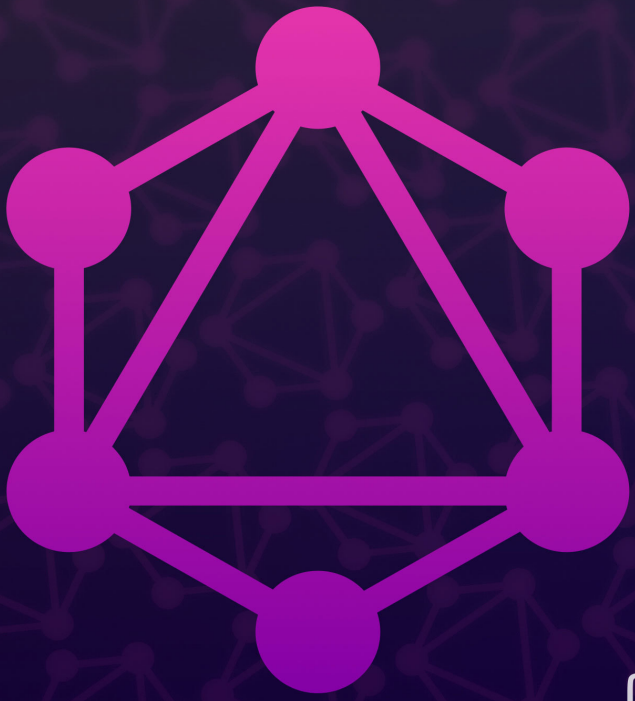
FULLSTACK GRAPHQL

Complete Guide to Building Servers and Clients in GraphQL



ROY DERKS

GAETANO CHECINSKI

newline

Fullstack GraphQL

The Complete Guide to Building GraphQL Clients and Servers

Written by Gaetano Checinski and Roy Derks
Edited by Nate Murray

Published by newline

Contents

What to Expect from this book

Fullstack GraphQL will give you a use-case driven, hands-on introduction to the GraphQL ecosystem. This book intends to present GraphQL to you in a problem-oriented and pragmatic fashion - and ultimately give you the confidence to use GraphQL in your everyday full-stack challenges.

This book shows both theoretical and conceptual aspects of GraphQL, including code examples book using React, Node.js, and Typescript to showcase how to use GraphQL to generate and render data.

The structure of this book is loosely inspired by the workflow of a Fullstack Engineer and will reveal GraphQL specific concepts through the life-cycle of a real-world app.

GraphQL is an opinionated query language, with an ecosystem that will change the way you write frontend and backend solutions.

Motivation

What is GraphQL

> "A query language for your API" - The GraphQL Foundation

GraphQL is a specification to query APIs, and provides a server-side runtime to execute those queries using a strongly typed system based on your data. Based on the data model, GraphQL can return you the data in exactly the same format as you request it in. It can be used language and framework agnostic, and connected to any database or storage engine.

In 2015 the specification for GraphQL was first publicly shared with the world by Facebook, after it was developed internally. To guarantee the continuation of GraphQL by the community, the project was moved into the GraphQL Foundation in 2019.

Why GraphQL?

Before we present you the problems that inspired the creation of GraphQL, let us have a look at this famous quote:

> "A language that doesn't affect the way you think about programming, is not worth knowing." - Alan Perlis

GraphQL changed the way how data was transferred between applications in a fixed format, to a new approach to dynamically transfer data between a "frontend" and a "backend". This allowed Facebook to tackle many problems with data fetching for mobile applications, without having to create a new REST API for every application.

As you will work through this book, we hope you won't just add GraphQL to your toolbox, but also develop a new way of thinking about data models, APIs and full-stack development.

The ecosystem surrounding GraphQL gives you the tools to start building and querying APIs, that are:

Usage Driven It encourages users to define queries that specify what data to fetch in a granular way.

Intuitive GraphQL delivers you only the data that you request, in the exact same format that you requested it in

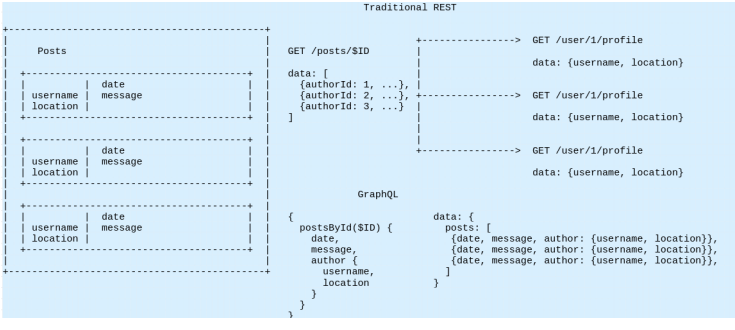
Self-descriptive GraphQLs schemas are strongly typed and define a strict contract between a query and its response. It makes GraphQL responses very predictable and also suitable for documentation.

GraphQL embodies many lessons learned from API design that enforces several best practices into one solution. As engineers, we're facing the challenge of not just building systems, but also evolving these systems to fit new requirements.

Usage driven and Intuitive

REST APIs were designed with specific use-cases in mind, making the development of "backend" and "frontend" loosely coupled or even independent.

The design of a REST API is often directly linked to the data model of the database that it's reading and mutating, making it a mere abstraction layer over this database.



In this example, we see a wire-frame of posts, how the underlying REST API, and a direct comparison with GraphQL.

When the REST API was created to serve the UI of a specific frontend application, the design of the REST API (and consequentially the data model of the database) should always match that UI. When the UI changes, the data flow and the data model of that database no longer match.

This can lead to a problem that often arises when you're used to working with REST APIs, that is called the n+1 problem. If you need to query multiple endpoints to collect enough data to render a page, then you are already facing the n+1 problem.

The n+1 problem

This problem describes the cascade of independent requests triggered by the lack of data in one response.

For the Developer Experience (DX) and performance reasons, it's therefore often desirable to fetch all of the data we need in one HTTP request to serve the UI. However, as your application grows in size, you find that you often end up creating a unique REST endpoint for every change in either the UI or the data model of the database.

For example, you created an endpoint that returns user information. This information was at first used to display a profile page but is also used to display a short overview of a user profile. Even later, you even use this information to display the user's avatar somewhere else.

This raises the following questions: 1) Should the backend grow with the requirements of each of it's clients? 2) Should the backend endpoints be changed as the client's requirements change? 3) Should the backend endpoints take configuration parameters to suit every client?

As we think through each of those options, we will notice that they all have their pros and cons.

Building a custom endpoint for a specific client allows you to fine-tune the backend for maximal performance. However, this will increase the number of endpoints to maintain.

Constantly adapting one endpoint to fit new requirements won't suffer the same issue but it will introduce a host of other questions: - How do we migrate old clients to the newer version? - Will we need to support both versions?

The last option combines the best of both worlds but introduces a level of indirection and requires a custom implementation.

GraphQL implements and standardizes this approach, as you'll discover in the first chapters of this book.

As you see, in GraphQL there is not only an intuitive mapping between queries and data but also it encodes the domain-specific language of your application.

Self-descriptive

REST is very minimalistic and does not enforce any types or schemas, and as a result, validation of input and output and documentation are complementary aspects of a REST API.

Consequently, validation and documentation is a maintenance burden. That is a potential source of bugs if the proper discipline isn't exhibited at all times. GraphQL has been designed with this in mind, leading to a more robust API and less overhead for developers.

All this is based on a GraphQL schema, a strongly typed and object-orientated representation of the data model for the application. The schema is used to both validate the requests and statically check the types of the operations and responses.

Being schema-driven also has an interesting side-effect, as the schema is always tied to the operations and responses and as a result, the schema is never out of date. We can always generate up-to-date documentation.

Other advantages

GraphQL provides many other advantages over a "traditional" approach for handling data flows between applications, as you'll discover in this book

Prerequisites

In this book we assumed that you have at least the following skills:

- basic JavaScript knowledge (working with functions, objects, and arrays)
- a basic React understanding (at least general idea of component based approach)
- some command line skill (you know how to run a command in terminal)

Here we mostly focus on specifics of using GraphQL with Node.js, TypeScript, and React.

The instructions we give in this book are very detailed, so if you lack some of the listed skills - you can still follow along with the tutorials and be just fine.

Running Code Examples

Each section has an example app shipped with it. You can download code examples from the same place where you purchased this book.

If you have any trouble finding or downloading the code examples, email us at us@fullstack.io[1].

In the beginning of each section you will find instructions of how to run the example app. In order to run the examples you need a terminal app and NodeJS installed on your machine.

Make sure you have NodeJS installed. Run `node -v`, it should output your current NodeJS version:

```
$ node -v
v10.19.0
```

> Anything between Node 10 through 14+ should be fine.

Here are instructions for installing NodeJS on different systems:

[1] mailto:us@fullstack.io

Windows

To work with examples in this book we recommend installing Cmder[2] as a terminal application.

We recommend installing node using nvm-windows[3]. Follow the installation instructions on the Github page.

Then run nvm to get the latest LTS version of NodeJS:

```
nvm install --lts
```

It will install the latest available LTS version.

Mac

Mac OS has a Terminal app installed by default. To launch it toggle Spotlight, search for terminal and press `Enter`.

Run the following command to install nvm[4]:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/inst\
all.sh | bash
```

Then run nvm to get the latest LTS version of NodeJS:

```
nvm install --lts
```

This command will also set the latest LTS version as default, so you should be all set.

If you face any issues follow the troubleshooting guide for Mac OS[5].

Linux

Most Linux distributions come with some terminal app provided by default. If you use Linux - you probably know how to launch terminal app.

Run the following command to install nvm[6]:

[2] https://cmder.net/
[3] https://github.com/coreybutler/nvm-windows
[4] https://github.com/nvm-sh/nvm
[5] https://github.com/nvm-sh/nvm#troubleshooting-on-macos
[6] https://github.com/nvm-sh/nvm

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/inst\
all.sh | bash
```

Then run nvm to get the latest LTS version of NodeJS:

```
nvm install --lts
```

In case of problems with installation follow the troubleshooting guide for Linux[7].

Code Blocks And Context

Code Block Numbering

In this book, we build example applications in steps. Every time we achieve a runnable state - we put it in a separate `step` folder.

```
01-first-app/
├── step1
├── step2
├── step3
... // other steps
```

If at some point in the chapter we achieve the state that we can run - we will tell you how to run the version of the app from the particular step.

Some files in that folders can have numbered suffixes with `*.example` word in the end:

```
src/AddNewItem0.tsx.example
```

If you see this - it means that we are building up to something bigger. You can jump to the file with same name but without suffix to see a completed version of it.

Here the completed file would be `src/AddNewItem.tsx`.

[7] https://github.com/nvm-sh/nvm#troubleshooting-on-linux

Reporting Issues

We've done our best to make sure that our instructions are correct and code samples don't contain errors. There is still a chance that you will encounter problems.

If you find a place where a concept isn't clear or you find an inaccuracy in our explanations or a bug in our code, you should leave a comment inline on newline.co.

> If you are reading this via PDF, did you know you can read all of newline's books online? You can either sync your purchases from your Gumroad account or read them via newline Pro[8]

You can also try emailing us[9]

In either case, we want to make sure that our book is precise and clear.

Getting Help

If you have any problems working through the code examples in this book, you should try:

- leaving a comment
- joining our Discord and asking or
- send us an email.

To make it easier for us to help you include the following information:

- What revision of the book are you referring to?
- What operating system are you on? (e.g. Mac OS X 10.13.2, Windows 95)
- Which chapter and which example project are you on?
- What were you trying to accomplish?
- What have you tried already?
- What output did you expect?
- What actually happened? (Including relevant log output.)

Ideally also provide a link to a git repository where we can reproduce an issue you are having.

[8] https://newline.co/pricing
[9] mailto:us@newline.co

Join Our Discord

We have a community Discord which you can join here[10] and join the #graphql channel

[10] https://newline.co/discord

Hello GraphQL - GraphiQL and Github's GraphQL Playground

In this chapter, we will be getting familiar with GraphQL by using Github's GraphQL explorer.

To get the most out of this chapter, we suggest you head over to The Github API Explorer[11] and follow along.

Below, we'll present code examples will guide you through GraphQL's intuitive, rich query language.

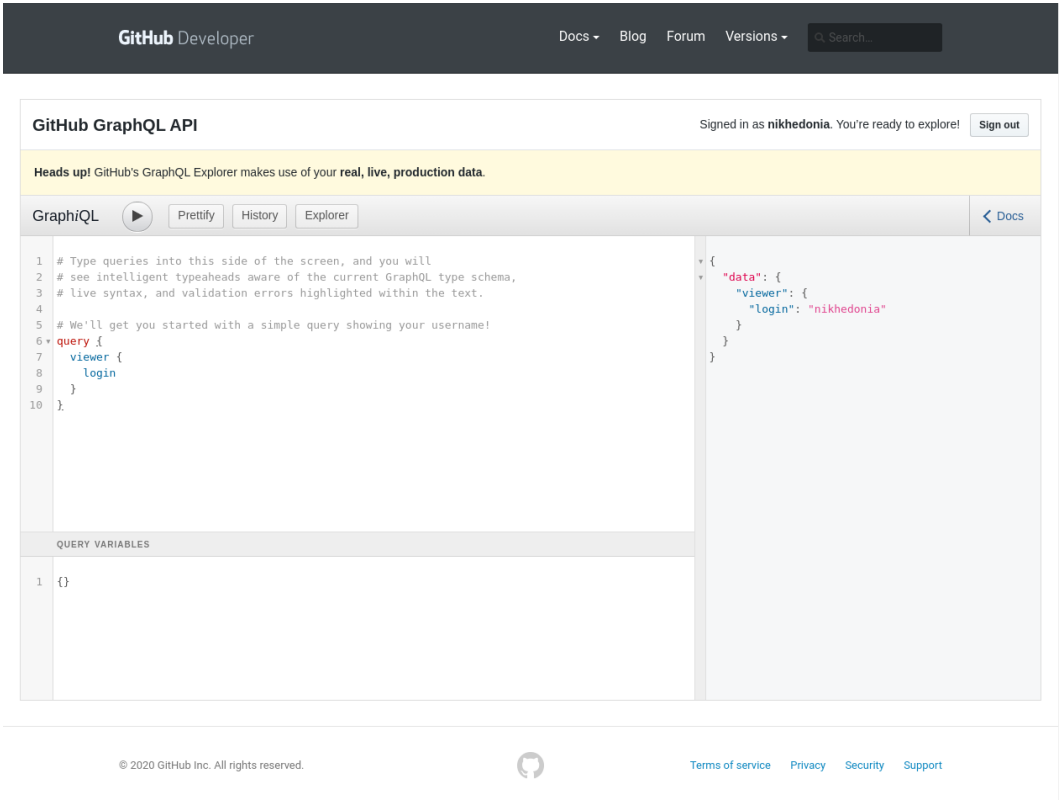
After this chapter you will be able to:

- Understand GraphQL schemas
- Read and write GraphQL Queries
- Explore APIs using the GraphQL Playground
- Send queries using `fetch` and `curl`

Enter GraphiQL

Once you navigate to The Github API Explorer and Sign-In you'll be welcomed with the online IDE that looks like this:

[11] https://developer.github.com/v4/explorer/



This is one of the most useful developer tools GraphQL offers and is known as GraphiQL - notice the *i* in the middle of that previous word. It stands for "interactive" and it's the GraphQL explorer.

It's your first stop when developing and exploring any GraphQL endpoint. GraphiQL provides an IDE experience with autocompletion, documentation and a simple query editor.

1. it syntax checks your queries
2. you can run queries and see the results
3. you can explore the types as you work

The documentation for exploring the types might be a little intimidating at first, but don't worry, we'll walk through how to make sense of it.

Our First Query

GraphQL queries look a little like JSON, here's our first query:

```
query {
  viewer {
    login
  }
}
```

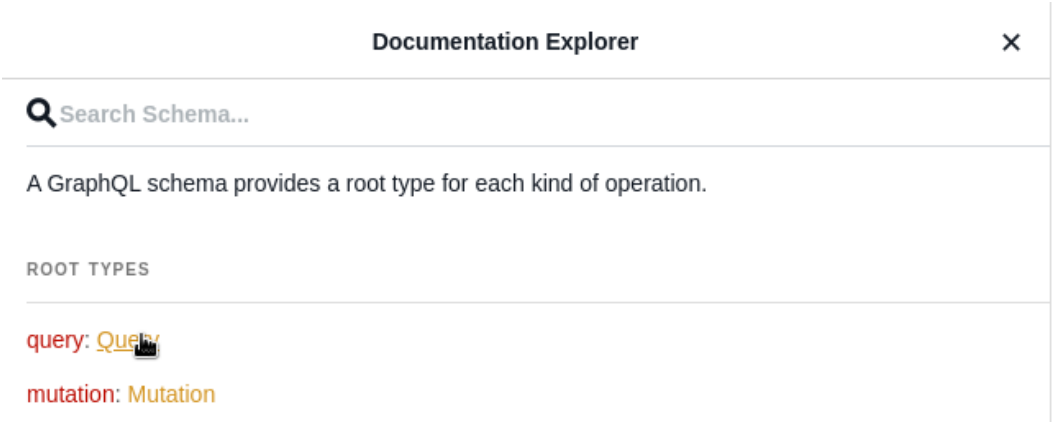
Above you'll see this isn't *quite* JSON though, we don't have any values! (Only keys). The response, however *is* in JSON. Let's take a look:

Here's the response for our query above:

```
{
  "data": {
    "viewer": {
      "login": "nikhedonia",
    }
  }
}
```

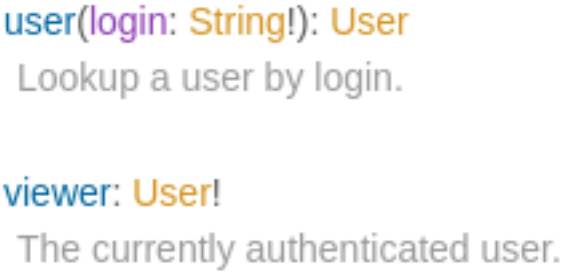
Notice that the shape of our request matches the shape of our response. And GraphQL fills in the *values* for the *keys* that we requested. This shape-matching is a powerful feature of GraphQL. But another powerful feature of GraphQL is it's **type system**.

Let's investigate what a viewer is and open the Documentation by hitting the "Docs" button on the right-hand side.



In GraphQL every field has a type associated and can be inspected in GraphiQL by clicking on it.

If you click on "Query" then you'll see all available fields that you can query - and among them you'll find `viewer: User!`



In English this means "the field **viewer** returns a **User**" and the exclamation mark means it will never return null.

The field `user` on the other hand is a *function* that takes a `login` of type `string` and returns a `User` but may return null if the User doesn't exist.

I think this is a noteworthy language choice: **all field declarations are considered optional by default**.

The documentation is generated directly from the schema. The GraphQL schema roughly looks like this:

```
# A user is an individual's account on GitHub ...
type User {
  login: String! # login is of type String. ! means it is never null
  email: String # email might be null
  # more fields...
}

type Query {
  user(login: String!): User
  viewer: User # The currently authenticated user
  # more fields...
}
```

Above we have a minimal schema with two user-defined types: `User` and `Query`. This is how to read it:

`User` has two fields: `login` and `email`, both of type `String`.

Because `login` is of type `String!` with an exclamation mark - it is may never be null. However, `email` may be null.

`Query` is a type with special importance: it marks the root and describes the starting point of any *query*. (Queries *read* data, but to write data we use what is called *mutations* - more on that below).

> This particular schema also exposes `viewer` as a field of `Query`, allowing us to reach into the data it holds. The `viewer` parameter here returns a `User` as well, but notice that there is no `login` parameter. That is because `viewer` returns *the current authenticated user*. More on this below.

Let's try to find users by their `login`:

```
query {
  user(login: "leebyron") {
    login
    name
    bio
  }
}
```

This query is asking for the `user` that has the `login` of `leebyron` and requests the fields `login`, `name` and `bio`. It returns:

```
{
  "data": {
    "user": {
      "login": "leebyron",
      "name": "Lee Byron",
      "bio": "I make things @robinhood, previously @facebook."
    }
  }
}
```

Lee Byron is one of the creators of GraphQL

Try putting in your own Github username and see what comes out!

If we pass a login that is not associated with any user (eg. `login: ""`) then it will return null:

```
{
  "data": {
    "user": null
  },
  "errors": [
    {
      "type": "NOT_FOUND",
      "path": [
        "user"
      ],
      "locations": [
        {
          "line": 7,
          "column": 3
        }
      ],
      "message": "Could not resolve to a User with the login of ''."
    }
  ]
}
```


Even more, the response contains also a descriptive error message with location information and error type.

This design allows GraphQL to return a valid query response even if only partial results - or no results - are available. We can query multiple fields and the server will return data even if some fields can't be "resolved". For example, we can ask for `viewer` in the same query:

```
query {
  viewer {
    login
  }
  user(login: "") {
    login
    name
    bio
  }
}
```

Which will return `viewer` even though no `user` could be found:

```
{
  "data": {
    "viewer": {
      "login": "nikhedonia"
    },
    "user": null
  },
  "errors": [
    # ...
  ]
}
```

Named Queries

It is highly recommended to name your queries. Although the name is optional, it is important for code generators and clients. Most clients use the name for caching purposes.

> For example, Apollo Client, a popular JavaScript client for GraphQL that we talk more about in future chapters, uses named queries. Also, tools like TypeScript type generation also uses named queries.
>
> So while names are technically optional, it's a good idea to use them.

So lets give our previous query a name:

```
query getUser {
  user(login: "leebryon") {
    login
    name
    bio
  }
}
```

The name goes right after `query` keyword - so above, `getUser` is the query name.

Note that we could name our query pretty much whatever we want. Instead of `getUser`, we could have named it `getLeeBryon`. The query name isn't "special" insofar as we can pick what we want.

However, the `user` *field* **is "special"** in that it's defined by the schema or GraphQL server. For example the following *would not* work:

```
query getUser {
  # this wont work, example of an invalid field `getUser`
  getUser(login: "leebryon") {
    login
    name
    bio
  }
}
```

Why does this not work? Because the inner `getUser` is not defined by the schema. More specifically, remember that our schema looks like this:

```
# A user is an individual's account on GitHub ...
type User {
  login: String! # login is of type String. ! means it is never null
  email: String # email might be null
  # more fields...
}

type Query {
  user(login: String!): User
  viewer: User # The currently authenticated user
  # more fields...
}
```

`getUser` is **not** a field on `Query`. The field is called `user`, and so that's what we have to use as well.

Variables

Oftentimes we want to specify a complex query and reuse it in different scenarios. Above we searched for the login `"leebryon"`, but we if we wanted to make a query that could search for *any* login? We use GraphQL *variables* for this.

Variables use a special syntax. Variable names start with `$` sign and must be defined in the query arguments. Let's have a look:

```
query getUser ($login: String!) {
  user(login: $login) {
    login
    name
    bio
  }
}
```

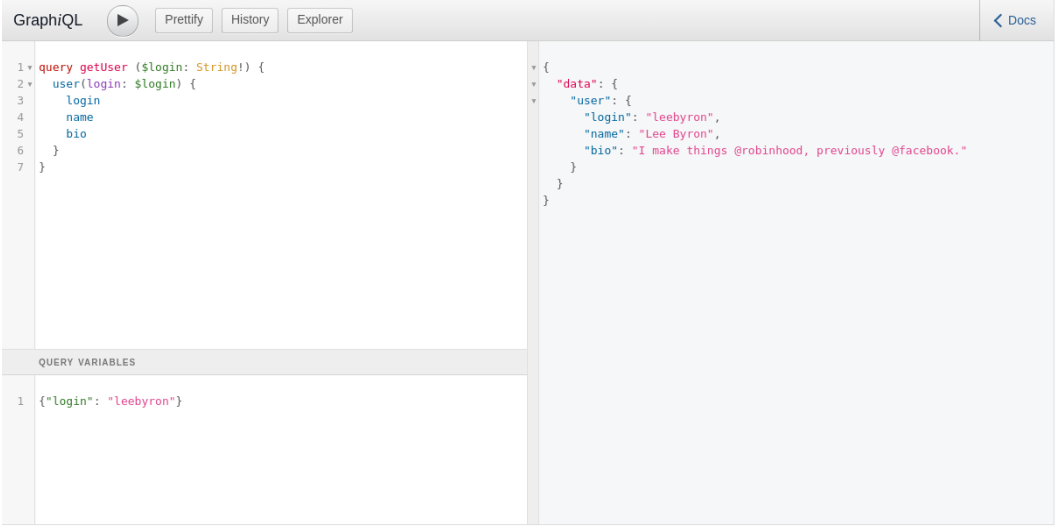
So here we have a named query `getUser` that defines one input variable `$login` which is a `String` and is mandatory (denoted by the `!`).

Look carefully: the `$login` variable is passed to the `login` *argument* of the `user` field. So how do we use this variable? Well it depends on the client.

When we're using GraphiQL we have to pass this parameter to make a valid query. In order to pass this variable we need to expand the "Variables" pane and then enter our variables in JSON like this:

```
{"login": "leebyron"}
```

And should look like this:



Again, try putting in your own Github username and see how the results change. Feel free to try different fields. Check: is your email address exposed by the Github API?

GraphQL Enforces Types and Correctness

GraphQL strictly enforces the correctness of inputs and will return an error if the variables don't satisfy the specification. If you provide the wrong type it will return an error.

For example, if we try to pass the number `5` like this:

```
{"login": 5}
```

then it will return:

```
{
  "errors": [
    {
      "extensions": {
        "value": 5,
        "problems": [
          {
            "path": [],
            "explanation": "Could not coerce value 5 to String"
          }
        ]
      },
      "locations": [
        {
          "line": 1,
          "column": 8
        }
      ],
      "message": "Variable Login of type String! was provided invalid v\
alue"
    }
  ]
}
```

GraphQL doesn't accept extra variables

For instance this will fail to execute this query:

```
query ($Login: String!, $NotUsed: String) {
  user(login: $login) {
    login
    name
    bio
  }
}
```

This is what the error will be:

```
{
  "errors": [
    {
      "path": [
        "query"
      ],
      "extensions": {
        "code": "variableNotUsed",
        "variableName": "Login"
      },
      "locations": [
        {
          "line": 1,
          "column": 1
        }
      ],
      "message": "Variable $Login is declared by  but not used"
    }
  ]
}
```

There's actually two problems with the above query:

1. `$Login` is capitalized so it isn't being used as the argument `$login`
2. `$NotUsed` is completely unused

Thankfully, GraphQL checks for that and will give us errors accordingly.

Multiple Queries and Renaming Result fields

As you noticed, the fields in the result object *match the fields in the query.*

What's cool is that GraphQL allows you to combine multiple queries into one. However, this can may lead to naming conflicts. To resolve this issue GraphQL allows re-mapping of result fields.

Let's try to query the data of two organizations at the same time:

```
query getOrganizations {
  facebook: organization(login: "facebook") {
    login
    description
  }

  microsoft: organization(login: "microsoft") {
    login
    description
  }
}
```

Here we're querying for **two** organizations at the same time. For each organization we get the `login` and the `description`, but because we're using aliases, **we have a different name for each organization in our results**. Such as this response:

```
{
  "data": {
    "facebook": {
      "login": "facebook",
      "description": "We are working to build community through open so\
urce technology."
    },
    "microsoft": {
      "login": "Microsoft",
      "description": "Open source, from Microsoft with love"
```

```
    }
  }
}
```

Notice that our results **do not** return the values in `organization`! We *renamed* them to `facebook` and `microsoft`.

This allows you to craft queries that return all data your application might need in one single request (but, of course, you have to take care when parsing the results).

Fragments

GraphQL queries can get quite large and there are a lot of good reasons to re-use "selections". For example, we might need to select a lot of fields for a company and we want to reuse that set of fields across our application.

Above, we had to specify that we wanted `login` and `description` for both companies, but wouldn't it be great if we could DRY-up that query?

It turns out we can, using *fragments*.

A `fragment` allows us to "factor out" the common fields of our query:

```
query getOrganizations {
  facebook: organization(login: "facebook") {
    ...commonOrgInfo
    # more fields
  }

  microsoft: organization(login: "microsoft") {
    ...commonOrgInfo
    # more fields
  }
}

fragment commonOrgInfo on Organization {
  login
```

```
  description
}
```

In this Example `commonOrgInfo` describes the fields of interest and can be *spread* into sub-queries of matching type.

> You can think of it as analogous to *spreading* in TypeScript or JavaScript objects.

Union Types and Interfaces

GraphQLs type-system supports two paradigms: inheritance and a thing called *discriminated unions*.

As a API consumer Union Types and Interfaces look identical and behave **almost** the same.

Both are usually used if one want's to reuse code or return an array of elements of different types.

Like in most programming languages we should favour inheritance via *Interfaces* if the types share some properties otherwise just use *Discriminated Unions*.

> Don't be scared by the terminology, we'll walk through each below.

Interfaces

`Interfaces` in GraphQL are very similar to interfaces from eg Java or class inheritance for languages like C++.

Lets take `Organizations` and `Users` as an example:

Both types are very similar and in fact they both extend the same interface - `RepositoryOwner`.

In GraphQL schema speak:

```
interface RepositoryOwner {
  login: String!
  avatarUrl: String
  # more fields ...
}

type Organization implements RepositoryOwner {
  login: String!
  avatarUrl: String
  description: String
  # more fields ...
}

type User implements RepositoryOwner {
  login: String!
  avatarUrl: String
  company: String
  bio: String
  # more fields ...
}
```

This reads as follows: `RepositoryOwner` is an interface and both `User`, and `Organization` implement that interface.

> Note that GraphQL requires you to repeat the fields of the `interface` in the `implementations`.

We can access the underlying type of an interface by matching on the type with `... on TYPE {...}` for instance:

```
query getOwner ($login : String) {
  repositoryOwner (login: $login) {
    ... on Organization {
      avatarUrl
      description
      name
    }

    ... on User {
      avatarUrl
      company
      bio
    }
  }
}
```

`... on Organization {...}` and `... on User {...}` are match expressions.

They conditionally extract the described fields from the object.

In this case we extract `{avatarUrl, description, name}` if the `repositoryOwner` is an `Organization` and `{avatarUrl, company, bio}` if the `repositoryOwner` is an `User`.

Furthermore we can take advantage of the built-in field `__typename` for introspection purposes.

`__typename` enables you to cleanly write logic that switches on the type. `__typename` is defined and available for all types not only for interfaces.

Additionally, all fields that were defined in the `repositoryOwner` interface can be refactored out of `Organization` and `User`. This leads to a more concise query:

```
query getOwner ($login : String) {
  repositoryOwner (login: $login) {
    __typename
    avatarUrl
    ... on Organization {
      description
      name
    }

    ... on User {
      company
      bio
    }
  }
}
```

We can even leverage re-mappings to unify over the differences and provide a consistent shape for our app. For instance image a component that displays a summary of a repository and it's owner:

```
query getProject($Name: String!, $Owner: String!) {
  repository(name: $Name, owner: $Owner) {
    name
    primaryLanguage {
      name
      color
    }
    pushedAt
    description
    owner {
      type: __typename
      login
      avatarUrl
      ... on Organization {
        name
        description
```

```
      }
      ... on User {
        name
        company
        description: bio # re-name bio
      }
    }
  }
}
```

Which would result in a response similar to:

```
{
  "data": {
    "repository": {
      "name": "graphql-spec",
      "primaryLanguage": {
        "name": "Shell",
        "color": "#89e051"
      },
      "pushedAt": "2019-06-20T09:23:42Z",
      "description": "GraphQL is a query language and execution engine \
tied to any backend service.",
      "owner": {
        "type": "Organization",
        "login": "graphql",
        "avatarUrl": "https://avatars0.githubusercontent.com/u/12972006\
?v=4",
        "name": "GraphQL",
        "description": ""
      }
    }
  }
}
```

Discriminated Unions

Consider the Github search:

If you perform a fulltext search for eg. "frontend" you might get back a repository, a code-snippet, an issue or an user. Each of them are very different in nature so how can graphql return a set of different things?

This is where *discriminated unions* come in. Defining a discriminated union types are denoted via the pipe `|` operator as follows:

```
type SearchNode= Repository | User | ...
```

The pipe `|` means `or`. So the above reads as follows:

> An element of type SearchNode is either a Repository OR User OR ...

As one might expect, discriminated Unions don't require it's types to have any relationship. This gives you great flexibility in designing your types as it doesn't force you to organize your types in a hierarchy.

This is how a complete search query looks like:

```
query findUrls($query: String!, $type: SearchType!){
  search(query: $query, type: $type) {
    nodes {
      __typename
      ... on Repository {
        url
      }
      ... on User {
        url
      }
    }
  }
}
```

`findUrls` finds urls of `Repositorys` or `Users` that match the query-string. The field nodes is of type `[SearchNode]` which is an array of `SearchNode`.

`SearchNode` is a discriminated union and defined as `type SearchNode = Repository | User | Project | # more types`

Even though `Repository` and `User` both have a `url` field, `url` cannot be hoisted into `SearchNode`.

Pagination

So far we only looked at examples where we can retrieve all data in one go. But what if we have more data then we would like to send over the wire in one chunk?

Like in traditional REST api we can build paginated apis that enforce multiple round trips.

Let's take an example of Github commit history API. It enables us to search the Git commit history *without cloning the repository.*

Let's look at a more complex query that involves many of the features we discussed above:

```
query GetCommits($owner: String!, $name: String!, $cursor: String) {
  repository(owner: $owner, name: $name) {
    defaultBranchRef {
      name
      target {
        ... on Commit { # match on type Commit
          history(first: 10, after: $cursor) {
            totalCount
            edges {
              cursor
              node {
                author {
                  email
                  user {
                    login
```

```
                }
              }
              message
              pushedDate
              deletions
              additions
              changedFiles
            }
          }
        }
      }
    }
  }
}
```

The Git commit history API is paginated and to navigate to each page we will use a *cursor*.

The query `GetCommits` selects the latest 10 commits that come after the provided *cursor*. A cursor is a pointer to a specific "node" - in this case a commit. If *$cursor* is undefined then the query will select the latest 10 commits of the history.

Notice the parameter `$cursor`. A cursor is like a pointer to a specific place. It is analogous - and a generalization - to an *offset* in an SQL query:

```
SELECT
  message
FROM History
WHERE repository = $<repository>
OFFSET $<after>
LIMIT $<first>
ORDER BY date
```

In this query we can access the first 10 entries with `first=10; after=0` and the next 10 with `first=10; after=10` and so on. A slightly more sophisticated pagination system could look like the following:

```
SELECT
  id as cursor
  message
FROM History
WHERE
  repository = $<repository>
  AND
  id < $<after>
LIMIT $<first>
ORDER BY id
```

`$<after>` and `$<first>` mirror here the role of `first` and `after` in GraphQL.

Here we can paginate in similar fashion but `after` would be the `cursor` of the last entry.

The expected way to use the API is as follows:

- edges returns an array of `CommitEdge`.
- each `CommitEdge` has a cursor and a node of type `Commit`
- in order to get the next 10 entries after the last entry, we take the cursor of the last edge and use it in the next query

For instance our first query might be using `{owner: "facebook", name: "react"}` which would return sth along the lines of:

```
{
  "data": {
    "repository": {
      "defaultBranchRef": {
        "name": "master",
        "target": {
          "history": {
            "totalCount": 11139,
            "edges": [
              {
```

```
      "cursor": "aabbcc", # first cursor
      "node": {
        message: "..."
        # more fields
      }
    },
    # 8 more entries
    {
      "cursor": "ddeeff", # last cursor
    }
# ...
```

and to get the next 10 pages we would use `{owner: "facebook", name: "react", cursor: "ddeeff"}`.

As often we only care about the first and last cursor there is also an alternative API using `pageInfo` and `nodes`:

```
query GetCommits($owner: String!, $name: String!, $cursor: String) {
  repository(owner: $owner, name: $name) {
    defaultBranchRef {
      name
      target {
        ... on Commit { # match on type Commit
          history(first: 10, after: $cursor) {
            totalCount
            pageInfo {
              startCursor
              endCursor
            }
            nodes { # this is an array of commits
              message: "..."
            }
# ...
```

Those are very common navigation pattern we will learn more about in the Pagination chapter.

Mutations

Mutations are a way of issuing changes in GraphQL. They use exactly same syntax as queries but semantically they have side-effects. Mutations may also return data and we can access the data like in ordinary queries.

> By analogy, if `query` is a `GET` request, `mutation` is a `POST`.
>
> (I'm not saying that you issue GraphQL queries using these specific HTTP methods, but rather they represent the read/write pair of GraphQL.)

For instance we can create a comment for a given `subjectId`:

```
mutation AddComment($subjectId: ID!, $message: String!) {
  addComment(input: {subjectId: $subjectId, body: $message}) {
    subject {
      id
      # more fields
    }
  }
}
```

The `AddComment` mutation does exactly what it says: it adds a comment with a specified *message* to an *subject*.

The **arguments** to `addComment` are the arguments to the mutation, that is what we're "writing".

The fields *inside* `addComment` are fields we're fetching **after the mutation has executed**, that is, what we're "reading".

We'll talk more about mutations later in the book, but for now, just know that the syntax is basically the same! You're just using `mutation` instead of `query` at the top and the expectation is that you're causing a side effect to happen.

GraphQL queries over HTTP - fetching with `fetch` and `curl`

So far, we've talked about how to issue GraphQL requests via the GraphiQL online experience. But what if we want to issue a GraphQL request more "directly" with `fetch` in JavaScript or the `curl` command-line tool?

> If you want to try to connect these to Github, you will need an auth token.
>
> You can create a GraphQL API key for Github if you navigate to https://github.com/settings/ ("Settings" > "Developer Settings" > "Personal access tokens").

The story is simple: it's just a POST request with the following `application/json` data.

```
{
  query: "query MyQuery { viewer {name} }",
  variables: {},
  operationName: "MyQuery"
}
```

The `operationName` here is the name of your query.

With curl, it would look like this:

```
curl \
  -X POST \
  -H "Authorization: Bearer $AUTH_TOKEN" \
  -H "Content-Type: application/json" \
  --data '...' \
  https://github.com/graphql
```

where the `...` are replaced by the JSON string from above.

With `fetch` it would look like this:

```
fetch("https://github.com/graphql", {
  method: "POST",
  credentials: "include",
  mode: "cors",
  headers: {
    Authorization: `Bearer ${AuthToken}`,
    "Content-Type": "application/json"
  },
  body: JSON.stringify(data)
});
```

Summary

GraphQL has a powerful and intuitive API for making queries and mutations, as you understand after reading this chapter. In this chapter, you learned how to about GraphQL schemas and how (possible) queries are defined in the schema.

From the schema, you can know how to read and write GraphQL queries, including named queries, and using variables and fragments to re-use field and queries.

Also, this chapter covered exploring queries using the GraphQL Playground and sending queries using `fetch` from your code or using `curl` from the command line.

In the next chapter, you get to apply this knowledge by building your very first GraphQL app with React.

Hello GraphQL from Node

In the previous chapter you had a first glimpse of the GraphQL Query Language. Now it's time to see how we can interact with a GraphQL endpoint from JavaScript, by using Github's GraphQL API that you've already familiarized yourself with in the previous chapter.

In this chapter we're going to create a super-basic GraphQL client in Node.js. In the next chapter, we'll do the same with React.

A GraphQL Hello World from Node

Our first goal will be to fetch your username from Github using GraphQL and print it to console.

We setup our project via `yarn init` and then install an HTTP client `yarn add superagent`. superagent[12] provides a nice, consistent API for making HTTP requests in both the browser and Node.js.

We create an `index.js` and add the following code:

code/2-hello-world-node/viewer.js

```
1  const { post } = require("superagent");
2
3  const query = `query {
4    viewer {
5      login
6    }
7  }`;
8
9  const token = process.env.GITHUB_TOKEN;
10 const headers = {
```

[12] https://visionmedia.github.io/superagent/

```
  "User-Agent": "superagent",
  Authorization: `Bearer ${token}`,
};

async function main() {
  const { body } = await post("https://api.github.com/graphql")
    .set(headers)
    .send({ query });

  console.log(body);
}

main();
```

Once saved we can run `GITHUB_TOKEN=<YOUR_TOKEN> yarn start`

> We gave instructions on how to acquire your `GITHUB_TOKEN` back in Chapter 1.

Now run the file like this:

```
node viewer.js
```

If its configured correctly, you should see a response like this:

```
{
  data: {
    viewer: {
      login: 'nikhedonia';
    }
  }
}
```

The code above, first sets the GraphQL query to:

```
const query = `query {
  viewer {
    login
  }
}`;
```

Github's GraphQL API only accepts requests via POST, a requires a defined `User-Agent` string and `Authorization` header. So we use superagent's `.set` method to set those headers.

Lastly we `.send` the query via POST.

Making Changes with Mutations

Now let's try doing something more useful: we'll create a CLI tool that can *create* repositories. In order to do this, we'll need to use *mutations*.

For this we change our query to the following mutation:

code/2-hello-world-node/create.js

```
const CREATE_REPO_QUERY = `mutation CreateRepository($name: String!, $b\
ody: String!) {
  createRepository(input: {name: $name, body: $body}) {
    id
  }
}`;
```

In the query above, `$name` and `$body` are variables and the values need to be provided in the `variables` field in the requests.

Note: While we could avoid using variables by inlining the strings for name and description. But it's not recommended for (unsanitized) user inputs.

This would allow evil users to create malicious strings that can change the intended behaviour - similar to SQL injection. Variables however make this difficult or even impossible. They are type-checked by the server and by some GraphQL clients, but ensure a separation between query and parameters.

Next, we read the arguments passed from the command line into the application:

code/2-hello-world-node/create.js

```
16   const [name, description] = process.argv.slice(2); // remove first tw\
17 o args because: argv[0] === 'node' and argv[1] === 'index'
18   const { body } = await post("https://api.github.com/graphql")
19     .set(header)
20     .send({
21       query: CREATE_REPO_QUERY,
22       variables: {
23         name,
24         description,
25       },
26     });
```

Now both variables can be passed as arguments into our CLI tool allowing us to create a repo via `GITHUB_TOKEN=<YOUR_TOKEN> node index myTestRepo "repo to test graphql"`

Summary

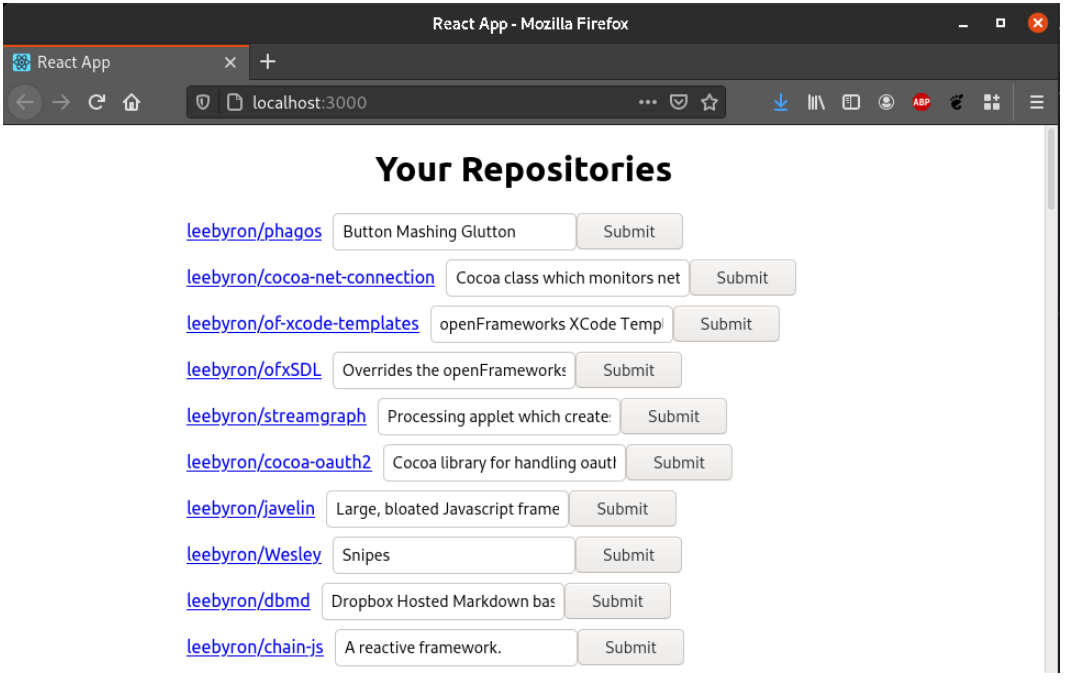
In some sense, that's all there is to it! You can interact with a GraphQL endpoint just like you might with any other REST endpoint. However, we will see soon more complex examples that leverage more features of GraphQL.

Next, let's take a look at how we'd create a basic example in the browser.

Hello GraphQL in the Browser

Now that we have a simple GraphQL app running in Node, in this chapter we'll build small repository management tool in React.

This is what we'll have built by the end of the chapter:



Repository Manager

The goal of this project is to get an intuitive understanding what it entails to work with GraphQL and React.

A GraphQL Hello World from Localhost

This should provide a smooth and natural transition to sophisticated clients like Apollo, which we cover in some of the next chapters.

For this chapter, we will build an app that displays a list of your repositories and allows you to quickly add or remove new repositories.

In order to get a project up and running we setup things with `create-react-app` and install some dependencies:

```
npx create-react-app repository-manager
cd repository-manager
yarn add superagent
```

The folder structure should look like:

```
repo-manager
├── build
│   ├── favicon.ico
│   └── manifest.json
├── package.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── README.md
├── src
│   ├── App.tsx
│   ├── index.css
│   ├── index.tsx
│   ├── logo.svg
│   ├── react-app-env.d.ts
│   ├── serviceWorker.ts
├── tsconfig.json
└── yarn.lock
```

Next we craft a query that gives us a list of repositories:

code/2-hello-world-cra/src/queries/fetch-repositories.js

```
2  import { post } from "superagent";
3
4  const FETCH_REPO_QUERY = `
5  query {
6    viewer {
7      repositories(first:100) {
8        nodes {
9          id
10         url
11         nameWithOwner
12         description
13       }
14     }
15   }
16 }`;
17
18 const TOKEN = process.env.REACT_APP_GITHUB_TOKEN;
19
20 export const fetchRepositories = () =>
21   post("https://api.github.com/graphql")
22     .set({
23       Authorization: `Bearer ${TOKEN}`,
24     })
25     .send({
26       query: FETCH_REPO_QUERY,
27     });
```

You'll notice that this code looks a lot like the request we made with `superagent` and Node.js in the previous chapter.

The `FETCH_REPO_QUERY` query retrieves a list of up to 100 repositories of the current user.

Then we export `fetchRepositores` function which will builds the post requests using the query string, your token and superagent.

The token is passed via `process.env` and by convention the React starter kit bakes all environment variables starting with `REACT_APP_*` into your app.

Environment variables defined in a `.env` are automatically loaded by a `create-react-app` app.

This also means that this variable will be compiled into your final JavaScript and be viewable by anyone who can view the code in their browser. The consequence is you need to be careful about what token you expose online and make sure you intend for it to be public (or not)

Putting the Query in the App

Now that we have this basic query and function to make the query, we can integrate it into our `App.js`.

In order to do this, we need to use React's `useEffect` hook and update the component via the `useState` callback:

code/2-hello-world-cra/src/AppQuickAndDirty.js

```
1  import React, { useState, useEffect } from "react";
2  import { fetchRepositories } from "./queries/fetch-repositories";
3
4  export default function App() {
5    const [repos, setRepos] = useState([]);
6    const [error, setError] = useState(null);
7
8    useEffect(() => {
9      fetchRepositories()
10       .then((x) => setRepos(x.data.viewer.repositories.nodes))
11       .catch((error) => setError(error));
12   }, []);
13
14   const repoList = repos.map((repo) => (
15     <div key={repo.id}>
```

```
16        <a href={repo.url}>{repo.nameWithOwner}</a>
17        <span>{repo.description}</span>
18      </div>
19    ));
20
21    return (
22      <div>
23        <h1> Your Repositories </h1>
24        {error ? <p>{error.toString()}</p> : repoList}
25      </div>
26    );
27  }
```

This Component uses `useEffect` to send off the Query **as soon as the component is rendered**.

We pass an empty dependency array to `useEffect` to ensure that it only fires *once*. But if the query depended on variables, then we would list the variables in the second argument to `useEffect` to ensure it fires the query if the variables change.

> If you're unfamilar with the semantics of `useEffect`, consider reading the documentation here[13].

When the query is resolved we update the list of repositories with the retrieved data on success and otherwise we set the error.

We then map the retrieved repositories to a visual representation using the returned `nameWithOwner` and `description`.

In the above code, we don't distinguish between loading and completed, but you could if you wanted to.

Creating a Custom Hook

Instead of hacking this into our app, let's first create a custom hook that encapsulates all this logic:

[13] https://reactjs.org/docs/hooks-effect.html

code/2-hello-world-cra/src/queries/fetch-repositories.js

```
29 export const useRepoQuery = (deps = []) => {
30   const [state, setState] = useState({
31     loading: true,
32     error: null,
33     data: null,
34   });
35
36   useEffect(() => {
37     fetchRepositories()
38       .then((x) => {
39         setState({
40           error: null,
41           loading: false,
42           data: x.body.data,
43         });
44       })
45       .catch((error) => setState({ loading: false, error, data: null })\
46 );
47   }, deps);
48
49   return state;
50 };
```

This is now a self-contained hook that handles the whole lifecycle of our GraphQL query and is ready to be consumed in our component.

> We could generalize this hook even more by refactoring out the query, variables and maybe even add some methods like `refetch`. If we would do this we would have implemented a part of the Apollo-client hooks you can find in the `@apollo/client` package.

Now we can use this hook in our `App.js`:

code/2-hello-world-cra/src/App.js

```
5  export default function App() {
6    const { loading, error, data } = useRepoQuery();
7
8    if (loading) {
9      return (
10       <div className="App">
11         <h2>Loading</h2>
12       </div>
13     );
14   }
15
16   if (error) {
17     return (
18       <div className="App">
19         <p>{error.toString()}</p>
20       </div>
21     );
22   }
23
24   const repos = data.viewer.repositories.nodes;
```

Making Updates with Mutations

Next, how about we add some functionality to update the description of a repository? The mutation-query to accomplish this would look like this:

code/2-hello-world-cra/src/queries/update-repo-description.js

```
const UPDATE_DESCRIPTION_QUERY = `
mutation updateDescription($repositoryId: ID!, $description: String!) {
  updateRepository(input: {repositoryId: $repositoryId, description: $d\
escription}) {
    repository {
      id
      description
    }
  }
}`;
```

This query takes a `repositoryID` and a `description` as variables.

We can take this query and wrap it into a function like we did with the `FETCH_REPOSITORY` query:

code/2-hello-world-cra/src/queries/update-repo-description.js

```
export const updateRepoDescription = (variables) =>
  post("https://api.github.com/graphql")
    .set({
      Authorization: `Bearer ${TOKEN}`,
    })
    .send({
      query: UPDATE_DESCRIPTION_QUERY,
      variables,
    });
```

Once this file is saved we can integrate this again into our App.

Handling User Input

As the new description is supposed to be editable by the user, we need an input component and a submission button.

We'll use `useRef` to get a reference of the input element and submit it via the buttons onClick event handler:

To handle this we can write a `Repo` component:

code/2-hello-world-cra/src/components/Repo.js

```
import React, { useRef, useState } from "react";
import { updateRepoDescription } from "../queries/update-repo-descripti\
on";

export const Repo = ({ repo }) => {
  const inputRef = useRef(null);
  const [isUpdating, setIsUpdating] = useState(false);

  const submit = () => {
    const repositoryId = repo.id;
    const description = inputRef.current.value;
    updateRepoDescription({ repositoryId, description })
      .then(() => {
        setIsUpdating(false);
      })
      .catch(() => {
        setIsUpdating(false);
        inputRef.current.value = repo.description;
      });
  };

  return (
    <div key={repo.id}>
      <a href={repo.url}>{repo.nameWithOwner}</a>
      <input ref={inputRef} defaultValue={repo.description} />
      <button disabled={isUpdating} onClick={submit}>
        {" "}
        Submit{" "}
      </button>
    </div>
```

```
    );
};
```

Our component allows editing via an input component and sends the update query on button click.

As the submission may take some time or even fail, we track the updating process by setting `isUpdating` as soon the query is submitted. This will trigger a re-render and prevent the user from submitting another update until this query is resolved.

In case of an error we reset the value of the input to the original description.

With this new component we can replace our old `repoList` implementation in App.js:

code/2-hello-world-cra/src/App.js

```
import React from "react";
import { Repo } from "./components/Repo";
import { useRepoQuery } from "./queries/fetch-repositories";

export default function App() {
  const { loading, error, data } = useRepoQuery();

  if (loading) {
    return (
      <div className="App">
        <h2>Loading</h2>
      </div>
    );
  }

  if (error) {
    return (
      <div className="App">
        <p>{error.toString()}</p>
      </div>
    );
  }
```

```
23
24    const repos = data.viewer.repositories.nodes;
25    const repoList = repos.map((repo) => <Repo repo={repo} key={repo.id} \
26  />);
27
28    return (
29      <div className="App">
30        <h1> Your Repositories </h1>
31        <div className="Content">
32          {error ? <p>{error.toString()}</p> : repoList}
33        </div>
34      </div>
35    );
36  }
```

Now we accomplished what we set out to do and you are now able to see your repos and edit their descriptions!

Summary

Now that you've learned how to build a basic React application that fetches the Github GraphQL API, you should try to experiment and add some additional features.

Furthermore there are still some refactoring opportunities left here, for example, we could generalize the `useRepoQuery` hook and maybe factor out our hardcoded credentials and URL of the endpoint. This would be a great exercise and you should give it a try!

There are a *ton* of different GraphQL clients out there and it can be hard to know which to use. Over the next few chapters, we'll be using a production-grade GraphQL client - `apollo-client` and it's react-bindings that resemble our `useRepoQuery` hook.

But before we do that, let's look at a few other options, shall we?

Picking a GraphQL Client

There are a bunch of GraphQL clients that exist. How do you know which one is right for you? I'm going to discuss some of your options, but let's cut to the chase:

If you're building a front-end web-app, I recommend you use Apollo Client. If you want to take my recommendation, you can just skip to the next chapter.

But for those of you who are making a years-long, fundamental architectural decision for your team - you might want to read below to get a lay of the land.

GraphQL is just JSON over HTTP (well, usually)

Because GraphQL servers are typically over HTTP, you don't *have* to use a client library at all. You can just use whatever HTTP request library you were already using and it will "work".

GraphQL is in this sense, more of a language/protocol pair than a specific library and it will work fine in any language where you can form queries and parse JSON - making it suitable for mobile apps, compiled clients, etc.

So as we covered in the first chapter, you can query a GraphQL server using `curl`,

```
{
  query: "query MyQuery { viewer {name} }",
  variables: {},
  operationName: "MyQuery"
}
```

The `operationName` here is the name of your query.

With curl, it would look like this:

```
curl \
  -X POST \
  -H "Authorization: Bearer $AUTH_TOKEN" \
  -H "Content-Type: application/json" \
  --data '...' \
  https://github.com/graphql
```

where the `...` is replaced by the JSON string from above

So why bother with a dedicated client?

Why You Might Want a Dedicated GraphQL Client

The big reason is because of **caching**.

Because our data is typed, and has a specific schema, intelligent clients can leverage that to cache data in clever, optimized ways.

For example, an object that is loaded by one query may *already have been cached* by a previous, completely different query. And in these cases, the client can read from the cache instead of fetching from the server - resulting in a snappier user experience.

There are lots of other nice features a dedicated client might have, too. Such as:

- Middleware
- Subscriptions
- File upload handling
- Server-side rendering support
- Authentication
- Optimistic updates
- Developer tools
- Testing libraries

All of the above can add up to a real-world advantage in programming time compared to just forming manual HTTP requests every time.

So What Are The Options?

So if you want to use a dedicated client, what are your options?

Here's a few:

- graphql-request[14] - a super minimal, no frills client
- urql[15] a lightweight, but batteries-included GraphQL client for React-like frameworks
- Relay Modern[16] a full-featured GraphQL client library which is great if you are Facebook
- Apollo Client[17] today's *de-facto* standard GraphQL library - works w/ React, Angular, Vue, etc.

Let's look a little closer at each one:

graphql-request

graphql-request[18] is a JavaScript library that, as it's namesake, makes it easy to make GraphQL requests.

Here's what the code looks like, taken straight from the documentation[19]

[14] https://github.com/prisma-labs/graphql-request
[15] https://github.com/FormidableLabs/urql
[16] https://github.com/facebook/relay
[17] https://github.com/apollographql/apollo-client
[18] https://github.com/prisma-labs/graphql-request
[19] https://github.com/prisma-labs/graphql-request

```
import { request, gql } from 'graphql-request'

const query = gql`
  {
    Movie(title: "Inception") {
      releaseDate
      actors {
        name
      }
    }
  }
`

request('https://api.graph.cool/simple/v1/movies', query)
  .then((data) => console.log(data))
```

What's good:

- it's lightweight
- it works with any JavaScript web framework
- it works in the browser or in node

The drawbacks:

It is super minimal - it doesn't have any caching or most of the advanced features you'd get out of any of the clients below.

Summary:

It's a great fit if you just want to make some quick requests without a lot of fuss.

urql

urql[20] gives you solid, lightweight choices in the default configuration, but is build to expand as you grow.

[20] https://github.com/FormidableLabs/urql

They have a nice page on the philosophy of urql[21] where they say:

> By default, we aim to provide features that allow you to build your app quickly with minimal configuration. urql is designed to be a client that grows with you. A/

Here's an example of making a query in a React app, taken from their documentation[22]

```
import { useQuery } from 'urql';
const TodosQuery = `
  query {
    todos {
      id
      title
    }
  }
`;
const Todos = () => {
  const [result, reexecuteQuery] = useQuery({
    query: TodosQuery,
  });
  const { data, fetching, error } = result;
  if (fetching) return <p>Loading...</p>;
  if (error) return <p>Oh no... {error.message}</p>;
  return (
    <ul>
      {data.todos.map(todo => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  );
};
```

[21] https://formidable.com/open-source/urql/docs/concepts/philosophy/
[22] https://formidable.com/open-source/urql/docs/basics/queries/

Above, you can see that they call the `useQuery` hook to load the `TodosQuery` and then use that result in the rendered JSX.

What's good:

- it's flexible
- it has great defaults
- it has a growing ecosystem of plugins
- it has a small bundle size[23] of just 23kb

The drawbacks:

As of writing, only supports React, Svelte, and Preact.

It's the "2nd place" option[24], with all of the benefits and drawbacks of that role.

Also, while it's lightweight to get started, you might find you're adding so many addins that it would have been easier to just use Apollo.

Summary:

A solid library, with critical community mass, from a great team. It's a good option.

Relay Modern

Relay Modern[25] a full-featured GraphQL client library and the original inspiration for Apollo.

What's good:

- it's the original inspirational GraphQL and React library
- it has real-world performance characteristics and has been tested at Facebook
- it's tightly integrated with React

The drawbacks:

[23] https://bundlephobia.com/result?p=urql@1.10.1

[24] https://npm-stat.com/charts.html?package=apollo-client&package=urql&package=react-relay&from=2019-10-11&to=2020-10-11

[25] https://relay.dev/

- it is highly opinionated in how you structure your components
- it is also opinionated on it's convention for pagination
- lower community adoption than you'd expect

Summary:

An inspirational, tightly-React-integrated library with many good characteristics, but generally not used as widely as you'd expect.

I imagine that Relay Modern will ship a version that gets broader community adoption. Almost everyone I know in the industry that *would* use Relay is using Apollo instead.

But if you're new to GraphQL and you're trying to decide between Relay and Apollo, and you don't know the difference, 9-times-out-of-10 you want Apollo.

Apollo Client

Like I mentioned above, Apollo Client[26] today's *de-facto* standard GraphQL library - works w/ React, Angular, Vue, etc.

What's good:

- it's batteries included
- everyone else uses it - so there's tutorials, plugins, etc.
- it supports more than React
- it just works

The drawbacks:

It's relatively "heavy". If you run into trouble, it's a bit of a black box.

Summary:

I use Apollo for newline and I've been incredibly happy. As often happens in open-source, there's ecosystem benefits with going with the leader.

There are more tutorials, bugs are fixed faster, and it's an overall pleasant experience.

Apollo also has tooling for building servers, so it's a bit of a one-stop-shop for GraphQL services.

[26] https://github.com/apollographql/apollo-client

What to Choose?

It's pretty obvious that I'm an Apollo fan. While there might be better choices for certain scenarios, if you're building a straightforward web app, my recommendation is Apollo client.

We'll be using Apollo client with React in the rest of this book. In the next chapter I'll show you how.

The Basics of Apollo Client and React

In a previous chapter, we connected our app to a GraphQL endpoint by just using the no-frills HTTP client `superagent`.

Now, we will be building upon our experience and see what a GraphQL client like `apollo-client` can do for us.

In this chapter we will be exploring the synergies between GraphQL and React by building a simple Github frontend.

In this chapter you will learn:

- how to setup and integrate Apollo Client with React
- how to use `apollo-client` with React
- the life cycle of a Query

  While we're using React in this section, it's important to know that Apollo also works with Angular and Vue.

Apollo Client

The ApolloClient manages the complexity of orchestrating all queries. It is responsible for scheduling, optimizing, caching and sending queries to a GraphQL-endpoint.

All the functionality is conveniently hidden behind it's `.query` method that manages and sends queries.

The Apollo Client has even a modular design, allowing you to mix and match the features you need.

But first let's migrate our previous app to Apollo. For this we need to install one dependency:

```
yarn add @apollo/client
```

This includes everything we need, the client, a caching solution and the React bindings.

We can now import and create our client:

code/3-hello-apollo-cra/src/client.js

```
import { ApolloClient, InMemoryCache, HttpLink } from "@apollo/client";

const token = process.env.REACT_APP_GITHUB_TOKEN;

const authorization = `Bearer ${token}`;

const cache = new InMemoryCache();
const link = new HttpLink({
  uri: "https://api.github.com/graphql",
  headers: {
    authorization,
  },
});

const client = new ApolloClient({
  link,
  cache,
});

export default client;
```

Here we see how to initialize the Apollo client: it takes the URL of the GraphQL endpoint, authorization header and the cache.

Let's test if everything is working correctly by firing a test query:

code/3-hello-apollo-cra/src/index.js

```
7  import { gql } from "@apollo/client";
8
9  const VIEWER_QUERY = gql`
10   query {
11     viewer {
12       login
13     }
14   }
15 `;
16
17 client.query({ query: VIEWER_QUERY }).then(console.log);
```

`client.query` takes here an object sends our query, and also handles optionally variables and bypassing of the cache or the network.

If everything worked out our `GET_VIEWER_QUERY` should succeed with a response of the shape `{ data, error: null, loading: false }`.

An important detail to note is that our query string is prefixed with a *graphql-tag* `gql`.

`gql` is a string-literal that parses the query string and returns a preprocessed representation (AST) of our query for faster execution.

Tools leverage the tag to generate wrappers, type definitions, and can even make our GraphQL apps faster and more secure with so-called *persisted queries*.

But for now, all you need to know is that `gql` needs to be used to create a valid GraphQL query for our Apollo client.

As an added bonus, all queries are cached by default meaning it will only send a request to the server if it can't find all data in it's cache.

You can easily verify this by sending the same query twice and then looking at the "Network" tab in your browser (You won't see a second request).

Getting Hooked on `useQuery`

`@apollo/client` ships with additional React bindings including:

- a set of hooks like `useQuery` and `useMutation` to send GraphQL queries
- `<ApolloProvider>` component to expose the apollo client via react context

Those hooks are recommended in a React app as they manage the complexities of managing the query and React-component lifecyle for you.

In order to use them, we need to wrap our App the following way:

code/3-hello-apollo-cra/src/index.js

```
19 ReactDOM.render(
20   <ApolloProvider client={client}>
21     <App />
22   </ApolloProvider>,
23   document.getElementById("root")
24 );
```

This is more or less the regular React rendering setup paired with the `ApolloProvider` that adds the Apollo client context to the React component tree.

From here on, now we can use `useQuery` and friends.

The API we designed in our homegrown GraphQL hook `useRepoQuery` provides a subset of `useQuery` and we can just replace:

code/3-hello-apollo-cra/src/queries/fetch-repositories.js

```
17 export const useRepoQuery = () => useQuery(FETCH_REPO_QUERY);
```

Getting hooked on `useMutation`

Next, we can do this for `updateDescription` mutation.

Mutations, however, take a special role in Apollo and need to be invoked using the `useMutation` hook.

As mutations may change data, `useMutation` will ensure that caches get invalidated, updated and components re-render.

`useMutation` takes a query and optional variables but unlike `useQuery` it **returns it's state** *and* a callback that needs to be called to fire the mutation query.

Consequently, the usage of `useUpdateDescription` is a bit different:

code/3-hello-apollo-cra/src/components/Repo.js

```
4  export const Repo = ({ repo }) => {
5    const inputRef = useRef(null);
6    const [update, { loading: isUpdating }] = useUpdateRepoDescription();
7
8    const submit = () => {
9      const repositoryId = repo.id;
10     const description = inputRef.current.value;
11     update({ repositoryId, description });
12   };
13
14   return (
15     <div key={repo.id}>
16       <a href={repo.url}>{repo.nameWithOwner}</a>
17       <input ref={inputRef} defaultValue={repo.description} />
18       <button disabled={isUpdating} onClick={submit}>
19         {" "}
20         Submit{" "}
21       </button>
22     </div>
23   );
24 };
```

The `update` function sends our mutation query and accepts additional query variables. Once called the loading status will change and data will contain the query result on completion.

How to use Apollo Client across your app

In some sense, it's that simple:

To load data, you fetch data using Queries - the Apollo cache helps ensure that you don't have to reload the same data twice.

To change data, you call your mutation callback function wherever you want perform an action (E.g. when you hit a button)

Of course, there is a lot of tooling required for a production app and we'll go into a lot more detail about that in the next chapter.

But first, let's talk about testing.

ApolloClient and Testing Components

Testing is a very important part of engineering - and `react-apollo` allows you to provide a mocked GraphQL client to your components.

We recommend writing tests in jest and the React testing library.

This is how we test the transition of of our `<App/>` from loading state to loaded state:

code/3-hello-apollo-cra/src/App.test.js

```
34 describe("<App/>", () => {
35   it("should transition from loading to loaded state", async () => {
36     const { getByText, getByDisplayValue } = render(
37       <MockedProvider mocks={mocks}>
38         <App />
39       </MockedProvider>
40     );
41
```

```
42      expect(getByText("Loading")).toBeTruthy();
43      expect(getByDisplayValue("repo description")).toBeFalsy();
44
45      await wait(); // wait untill loading completed
46
47      expect(getByText("Loading")).toBeFalsy();
48      expect(getByDisplayValue("repo description")).toBeTruthy();
49    });
50  });
```

Here's how it works in detail.

Setting up the Mocked Provider

First we render our component with a mocked Provider:

code/3-hello-apollo-cra/src/App.test.js

```
36      const { getByText, getByDisplayValue } = render(
37        <MockedProvider mocks={mocks}>
38          <App />
39        </MockedProvider>
40      );
```

The `MockedProvider` exposes a mocked apollo client with a set of defined `mocks` - a list of responses the client is supposed to return for a given query.

In our case we just need to mock `FETCH_REPO_QUERY`:

code/3-hello-apollo-cra/src/App.test.js

```
const mocks = [
  {
    request: {
      query: FETCH_REPO_QUERY,
    },
    result: {
      data: {
        viewer: {
          repositories: {
            nodes: [
              {
                id: "testId",
                nameWithOwner: "test/test",
                url: "https://test.com",
                description: "repo description",
              },
            ],
          },
        },
      },
    },
  },
];
```

Mocks are a list of rules where `request` is a pattern and `result` the response that should be returned if the given pattern is found.

You can use `error` as a response and `variables` inside `request` to handle more complex cases.

Testing State

The render function returns several methods to find specific elements form the rendered dom tree.

We use `getByText` to find a component by it's text content and `getByDisplayValue` to test the value of the `<input/>` field that holds our description.

code/3-hello-apollo-cra/src/App.test.js

```
43      expect(getByDisplayValue("repo description")).toBeFalsy();
```

We then assert that:

1. while loading, no description is found and
2. text `Loading` somewhere in the application.

code/3-hello-apollo-cra/src/App.test.js

```
35    it("should transition from loading to loaded state", async () => {
36      const { getByText, getByDisplayValue } = render(
37        <MockedProvider mocks={mocks}>
38          <App />
39        </MockedProvider>
40      );
41
42      expect(getByText("Loading")).toBeTruthy();
43      expect(getByDisplayValue("repo description")).toBeFalsy();
44
45      await wait(); // wait untill loading completed
46
47      expect(getByText("Loading")).toBeFalsy();
48      expect(getByDisplayValue("repo description")).toBeTruthy();
49    });
```

One could be more specific about where precisely the text appears but the more precise you are, the more likely you are to break the test when improving the components.

The goal is to write tests that assert the existence of things without depending too much on the implementation of the actual component.

Transition from loading to loaded State

The mocked Apollo client always returns the query in the loading state and, afterwards, transitions to the success state on the next event loop tick.

We can `await` a timeout `await wait(0)` to suspend the execution of our test until the next tick. Here's the implementation of `wait`:

code/3-hello-apollo-cra/src/App.test.js

```
8 const wait = (ms = 0) => act(() => new Promise((done) => setTimeout(don\
9 e, ms)));
```

> Note: as component state changes during this process and hooks are involved we need to wrap it in `act`.

Remember

- use the `@apollo/client` package to import Apollo Client
- `useQuery` is a hook that allows us to make queries
- `useMutation` is the that sends mutations
- Wrap your app in `ApolloProvider`, in order to make both of the above hooks work
- Test your components by *mocking* the `ApolloProvider`

The next chapter of this book will show how you can automatically generate Apollo client components from your GraphQL schema by using TypeScript.

Building a TypeSafe GraphQL React Client App - Part 1

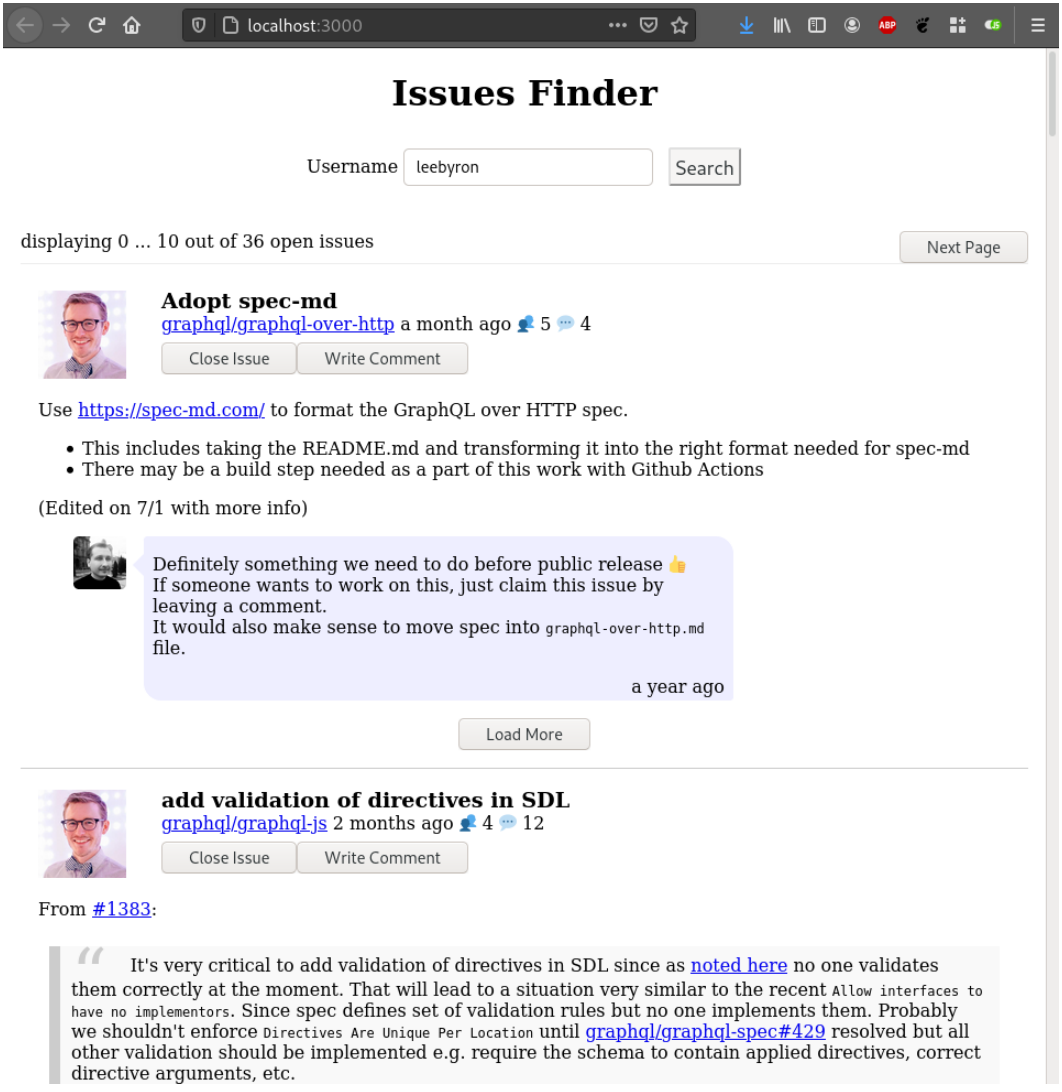
What are we building?

Our next app will piggyback on the Github's GraphQL endpoint to build a simple issue search app.

The app will allow you to quickly see a user's most recent engagements and allow you to comment and close issues.

In this chapter you'll learn:

- How to setup and structure your React and GraphQL
- How to build small stateful components using React hooks and Apollo



IssueFinder

Tooling and Project Structure

We will be using `create-react-app` `tsc` and `graphql-codegen` to setup our workspace. `graphql-codegen` will generate Typescript types for us and some handy wrappers to

simplify the usage of several Apollo methods.

So to get started run:

```
npx create-react-app my-app --template cra-template-fullstack-apollo
```

This installs all dependencies needed to build and test a React + TypeScript + Apollo app.

> If you like, you can also find our completed version of this code in the code downloads.
>
> For advanced users, we also include a copy of the create-react-app template in the code downloads under `code/cra-template-fullstack-apollo`

The app will have the following project structure:

```
github-client
├── build
|   ├── favicon.ico
|   └── manifest.json
├── package.json
├── public
|   ├── favicon.ico
|   ├── index.html
|   └── manifest.json
├── README.md
├── src
|   ├── hooks
|   |   └── index.tsx
|   ├── queries
|   |   └── viewer.tsx
|   ├── App.tsx
|   ├── index.css
|   ├── index.tsx
|   ├── logo.svg
|   ├── react-app-env.d.ts
```

```
|   ├── serviceWorker.ts
|   ├── graphql.tsx
├── tsconfig.json
├── codegen.yml
└── yarn.lock
```

This is a minimal sample application that we use as a starting point.

From now on you can run `yarn start` to start a Webpack-powered development server with hot-reload.

> This server will server your app on http://localhost:3000 and will update the app every time you modify your source code

TypeScript and GraphQL Types - There is a difference

One of the biggest features of using TypeScript and GraphQL together is that you can generate code for strong types from GraphQL.

This is incredibly powerful. Why? Because it means we can have **TypeScript types that match our GraphQL types and queries**.

This is an interesting nuance that might take a minute to understand: **TypeScript has a set of types** and **GraphQL has a set of types** – *but these are different things.*

GraphQL and TypeScript are two different languages with slightly different semantics.

But we can use tooling to generate TypeScript types from our GraphQL *schema* and *queries.*

Below, we'll look more closely at how to do this and how it works in practice.

The first thing that we need to do is **generate a list of TypeScript types** from our GraphQL schema.

"But *what* schema?" you may ask? Right now, we're going to use the Github GraphQL API schema.

We've pre-configured this for you and you can see it in the `codegen.yml` file.

code/4-issue-finder/codegen.yml

```
overwrite: true
schema:
  - "https://api.github.com/graphql":
      headers:
        Authorization: Bearer 42a6cb9f7bc5f157b7cdb8c1014b2ac14551a10c
documents:
  - src/**/*.tsx
generates:
  src/graphql.tsx:
    plugins:
      - "typescript"
      - "typescript-operations"
      - "typescript-react-apollo"
    config:
      withHooks: true
```

Notice that in the URL we've specified a `schema` URL. So that's the schema we'll read from to generate the types.

Generating Types with graphql-codegen

This project is configured with `graphql-codegen` that can fetch schemas from remote servers and **generate Typescript typings and Apollo helper functions**.

You can start the process with `yarn generate` or `graphql-codegen --config codegen.yml`.

If you dare to open the file, you'll see a set of definitions like:

code/4-issue-finder/src/graphql.tsx

```
18374 /** A user is an individual's account on GitHub that owns repositories \
18375 and can make new content. */
18376 export type User = Node &
18377   Actor &
18378   PackageOwner &
18379   ProjectOwner &
18380   RepositoryOwner &
18381   UniformResourceLocatable &
18382   ProfileOwner &
18383   Sponsorable & {
18384     __typename?: "User";
18385     /** Determine if this repository owner has any items that can be pi\
18386 nned to their profile. */
18387     anyPinnableItems: Scalars["Boolean"];
18388     /** A URL pointing to the user's public avatar. */
18389     avatarUrl: Scalars["URI"];
18390     /** The user's public profile bio. */
18391     bio?: Maybe<Scalars["String"]>;
```

This is an excerpt of `src/graphql.tsx` describing an User type.

`Node & Actor & ...` means a User contains (think of inheritance) the same fields as the Node, Actor (etc...) types.

Additionally, it has several fields like:

- a `__typename` set to `User`
- a `avatarUrl` of type URI (which is a string)
- an optional field `bio` of type string `Maybe<Scalars["String"]>` which is a string or `null`

But the key idea here is that we **generated a *TypeScript* file** that contains all of the types from our GraphQL server's schema.

Github has a `User` type in their GraphQL schema, and so here we've generated a `User` type **in TypeScript**.

Generating Types for Queries

The sample project also contains this query:

code/4-issue-finder/src/queries/get-viewer.tsx

```
import gql from "graphql-tag";
export default gql`
  query GetViewer {
    viewer {
      login
      name
      avatarUrl
    }
  }
`;
```

This query displays the information of the Viewer - more precisely the Owner of the Github Authorization Token - which is of type `User`.

One of the things `graphql-codegen` will do is **generate types for your queries**.

Why? Because it lets you have strict typings **that even match your queries**.

In the previous section we noted that we generated types for the `User` object. But what if we have a query that only pulls a few fields? If we typed on `User` but didn't query the "email" field, for instance, then our types would be wrong – the email would be null even though the type might say it's non-nullable.

Furthermore, the query-itself can be of a complex shape. We might query a `User` alongside a `Repository` and an `Issue` with subfields of each. It would be a massive pain to hand-create these types.

And so `graphql-codegen` generates these types for us. Let's take a look.

code/4-issue-finder/src/graphql.tsx

```
19528 export type GetViewerQuery = { __typename?: "Query" } & {
19529   viewer: { __typename?: "User" } & Pick<User, "login" | "name" | "avat\
19530 arUrl">;
19531 };
```

So this type, while it's not the easiest to read, matches our `GetViewerQuery` above.

Generating Helper Functions for Apollo

Another cool feature of `graphql-codegen` is that it will also generate helper functions we can use with Apollo. For example, it will generate a type-safe hook we can use in our React components.

Somewhere in `src/graphql.tsx` you'll also find:

code/4-issue-finder/src/graphql.tsx

```
20285 /**
20286  * __useGetViewerQuery__
20287  *
20288  * To run a query within a React component, call `useGetViewerQuery` an\
20289 d pass it any options that fit your needs.
20290  * When your component renders, `useGetViewerQuery` returns an object f\
20291 rom Apollo Client that contains loading, error, and data properties
20292  * you can use to render your UI.
20293  *
20294  * @param baseOptions options that will be passed into the query, suppo\
20295 rted options are listed on: https://www.apollographql.com/docs/react/ap\
20296 i/react-hooks/#options;
20297  *
20298  * @example
20299  * const { data, loading, error } = useGetViewerQuery({
20300  *   variables: {
20301  *   },
```

```
20302  * });
20303  */
20304 export function useGetViewerQuery(
20305   baseOptions?: Apollo.QueryHookOptions<GetViewerQuery, GetViewerQueryV\
20306 ariables>
20307 ) {
20308   return Apollo.useQuery<GetViewerQuery, GetViewerQueryVariables>(
20309     GetViewerDocument,
20310     baseOptions
20311   );
20312 }
```

This is a type-safe wrapper around `useQuery` and it even includes a documentation which will show up in VSCode.

Note that it also has the same API as the hooks built ourselves in the previous chapters.

Building the Issue Finder

First we design a query to find issues to a given user:

code/4-issue-finder/src/queries/get-issues.minimal.tsx

```
1  import gql from "graphql-tag";
2  export default gql`
3    query GetIssuesMinimal($login: String!) {
4      user(login: $login) {
5        id
6        login
7        avatarUrl
8        issues(
9          first: 10
10         orderBy: { field: UPDATED_AT, direction: DESC }
11         states: [OPEN]
12       ) {
```

```
13          totalCount
14          nodes {
15            id
16            url
17            number
18            author {
19              avatarUrl
20              login
21            }
22            repository {
23              nameWithOwner
24              url
25            }
26            title
27            bodyHTML
28            updatedAt
29          }
30        }
31      }
32    }
33  `;
```

This query finds the latest 10 `OPEN` issues sorted by `UPDATED_AT` field.

Notice that within the `issues` connection we have a `nodes` field. This is a common GraphQL convention. In this case, `nodes` is a list of issues.

You'll often see GraphQL servers use this `nodes` type of connection because it allows us to pass parameters (to the relationship `issues`) and receive metadata (such as the `totalCount` of records, pagination, etc.)

Once we've added this new query, we can call `yarn generate` to update `src/graphql.tsx`. and use the `useGetIssuesMinimalQuery` wrapper:

code/4-issue-finder/src/App.1.tsx

```tsx
import "./App.css";
import React from "react";
import { useGetIssuesMinimalQuery } from "./graphql";

type IssueFinderProps = {
  login: string;
};

const IssueFinder = (props: IssueFinderProps) => {
  const { login } = props;
  const { data, loading, error } = useGetIssuesMinimalQuery({
    variables: {
      login
    }
  });

  if (loading) {
    return <div> Loading </div>;
  }

  if (error) {
    return <div> Error occurred {error.toString()} </div>;
  }

  return (
    <div>
      <img src={data?.user?.avatarUrl} alt={data?.user?.login} />
      <div>
        login: {data?.user?.login}
        has {data?.user?.issues?.totalCount} open issues
      </div>
    </div>
  );
};

```

```
36 export default () => {
37   return <IssueFinder login="leebyron" />;
38 };
```

This component fires the `GetIssueMinimalQuery` and re-renders as the data is fetched.

For now, we only display the user's `login` and `totalCount` of issues - but this time it's type-checked!

Notice that we use TypeScript's optional chaining operator `?.`, which allows accessing properties of potentially null objects in a terse manner.

If you're unfamiliar with this syntax note that:

`data?.user` translates to `data ? data.user : null`

If you were to inspect the types of the `useGetIssuesMinimalQuery`

For reference the hook `useGetIssuesMinimalQuery` returns a result of type `GetIssuesMinimalQuer` and data is of type `GetIssues`.

Furthermore `useGetIssuesMinimalQuery` also takes variables of type `GetIssuesMinimalQueryVar` which is in our case an object with a github `login`.

Creating the Search Component

Next we can create an input component that drives the search:

code/4-issue-finder/src/components/SearchForm.tsx

```
3  export interface SearchFormProps {
4    login: string;
5    setLogin: (value: string) => void;
6  }
7
8  const SearchForm = ({ login, setLogin }: SearchFormProps) => {
9    const inputRef = useRef<HTMLInputElement>(null);
10
11   const onClick = () => {
12     setLogin((inputRef.current || { value: login }).value);
13   };
14
15   return (
16     <div className="SearchForm">
17       <label>
18         Username
19         <input
20           type="text"
21           ref={inputRef}
22           defaultValue={login}
23           placeholder="your login"
24         />
25       </label>
26       <button onClick={onClick}> Search </button>
27     </div>
28   );
29 };
```

Now we are able to type a login and search by clicking on the button.

The `onClick` method will trigger the event handler which, in turn, will read the current input value from the `inputRef` and trigger a re-render with the new login by calling `setLogin`.

If you have been running the development server via `yarn start` and typing along

you should be seeing your browser updating the view as you've been extending the React components.

Visualizing Issues

Now we can take it further and visualize all issues:

code/4-issue-finder/src/components/IssueCard.simple.tsx

```
import React from "react";
import { Issue } from "../graphql";

export const IssueCard = (issue: Issue) => (
  <div key={issue?.id} className="IssueCard">
    <div className="IssueCardHeader">
      <img src={issue?.author?.avatarUrl} />
      <div>
        {" "}
        <h3> {issue?.title} </h3>{" "}
      </div>
    </div>

    <div className="IssueContent">
      <h1> {issue?.title} </h1>
      <p dangerouslySetInnerHTML={{ __html: issue?.bodyHTML }} />
    </div>
  </div>
);
```

`IssueCard` is a function that maps `Issues` to React components.

Notice that the `Issue` type has been conveniently generated by `graphql-codegen`! Remember, we generated this earlier by querying the Github GraphQL endpoint and placed the code in `graphql.tsx`

Now we just need to import and consume this component in our `IssueFinder`.

code/4-issue-finder/src/App.2.tsx

```
26   const issues = (data?.user?.issues?.nodes || []) as Issue[];
27
28   const issuesList = issues.map((x: Issue) => <IssueCard key={x.id} {..\
29 .x} />);
30
31   return (
32     <div>
33       <div className="Results">{issuesList}</div>
34     </div>
35   );
36 };
```

This code drills down into the list of issues and renders each of them.

Notice how well the result of the query maps to our component hierarchy - this is part of the beauty of GraphQL.

Pagination with cursors

Often we have more data than fits on one page. So we need a way to keep track of what page we're on and how to get the next page of results.

Github's GraphQL Server uses *cursor* based pagination where we specify how many items to fetch `before` or `after` a specific `cursor`. Here is the important part of the new query:

code/4-issue-finder/src/queries/get-issues.pagination.tsx

```
3   query GetIssuesMinimalWithPagination(
4     $login: String!
5     $before: String
6     $after: String
7   ) {
8     user(login: $login) {
9       id
10      login
11      avatarUrl
12      issues(
13        first: 10
14        before: $before
15        after: $after
16        orderBy: { field: UPDATED_AT, direction: DESC }
17        states: [OPEN]
18      ) {
19        pageInfo {
20          hasNextPage
21          hasPreviousPage
22          startCursor
23          endCursor
24        }
25        totalCount
```

Note the newly added `pageInfo` field, it contains all necessary information for pagination: - `hasNextPage`, `hasPreviousPage` tell us the available pagination directions and thus whether we should render Previous and Next Page buttons - `startCursor` and `endCursor` are cursor representing the beginning and end of our currently displayed list.

In other words:

If we want to paginate forwards we query the first 10 `after` the `endCursor` - the cursor of the last node in our list.

If we want to paginate backwards we would query the first 10 `before` the `startCursor` the cursor of the first node in our list

Tracking our cursorState

For this we need to track the `cursorState` and feed it to our query:

code/4-issue-finder/src/App.4.tsx

```
12 type CursorState = {
13   before?: string | null;
14   after?: string | null;
15 };
16
17 const IssueFinder = (props: IssueFinderProps) => {
18   const { login } = props;
19   const [cursorState, setCursor] = useState<CursorState>({});
20   const { data, loading, error } = useGetIssuesMinimalWithPaginationQue\
21 ry({
22     variables: {
23       login,
24       ...cursorState,
25     },
26   });
```

Our `CursorState` is an object containing nothing or either `before` cursor or an `after` cursor.

Then the cursorState is spread into the query variables to fetch the next or previous nodes.

In order to modify the `cursorState` we call `setCursor` when a previous or next buttons are pressed:

code/4-issue-finder/src/App.4.tsx

```
38   const onPrevious = () => setCursor({ before: pageInfo?.startCursor });
39   const onNext = () => setCursor({ after: pageInfo?.endCursor });
```

The `onPrevious` and `onNext` callback set `before` and `after` as described above. Lastly we need to render buttons that invoke those callbacks when clicked:

code/4-issue-finder/src/components/PaginationControls.tsx

```
10 const PaginationControls = ({
11   hasPreviousPage = false,
12   hasNextPage = false,
13   onPrevious = () => {},
14   onNext = () => {},
15 }: PaginationControlsProps) => (
16   <div>
17     {hasPreviousPage && (
18       <button
19         onClick={() => {
20           onPrevious();
21           window.scrollTo(0, 0);
22         }}
23       >
24         Previous Page
25       </button>
26     )}
27     {hasNextPage && (
28       <button
29         onClick={() => {
30           onNext();
31           window.scrollTo(0, 0);
32         }}
33       >
34         Next Page
35       </button>
36     )}
```

```
37    </div>
38  );
```

Here our Pagination Component conditionally renders Previous and Next buttons. We also scroll to top to ensure the user sees the beginning of the newly rendered list.

`PaginationControls` will be a React component that renders a previous and next button.

Lastly we need to render the component and test our newly added feature:

code/4-issue-finder/src/App.4.tsx

```
41    const issuesList = issues.map((x: Issue) => <IssueCard key={x.id} {..\
42  .x} />);
43    const Controls = (
44      <PaginationControls {...pageInfo} onPrevious={onPrevious} onNext={o\
45  nNext} />
46    );
47
48    return (
49      <div className="Results">
50        <div>{issuesList}</div>
51        {Controls}
52      </div>
53    );
```

Now we are able to paginate forwards and backwards. This is a simple and bulletproof approach suitable for deep-linking and infinite scrolling applications.

The cursors would make a great unique URL and the page object contains everything needed to paginate in either direction. Furthermore Apollo caches each query, making it fast to navigate through pages we have already visited.

Improved Caching during Pagination

If you navigate forward, you call issues with `{after: endCursor}` but if you navigate backwards you call it with `{before: startCursor}`. Even if both would point to the same data sources, Apollo has no way of knowing that they lead to the same results.

If we remember the previous endCursor we can avoid this issue and speedup the backwards pagination.

For this we can create a small custom hook:

code/4-issue-finder/src/hooks/index.ts

```
5  export const useCursorPaginator = () => {
6    const [cursors, setCursors] = useState<(string | null)[]>([]);
7
8    return {
9      current: cursors[0],
10     cursors,
11     next: (next: string | null) => {
12       setCursors([next, ...cursors]);
13     },
14     previous: (n = 1) => {
15       setCursors(cursors.slice(n));
16     },
17   };
18 };
```

This hook returns `next(endCursor)` and `previous()` functions that either add or remove a cursor from a list of cursors.

If `next` or `previous` is called then it changes the current cursor that is supposed to be passed as `$after`. This ensures that we don't need to call our query with `{before: startCursor}` and instead use a value we know we have in our Apollo cache making backwards pagination instantaneous.

Note we can just remove the top cursor during backward navigation as that cursor will be returned by our query again it's the endCursor of the previous page.

We can connect this by passing those callbacks into our `onPrevious` and `onNext` callbacks:

code/4-issue-finder/src/App.5.tsx

```
18   const { current, cursors, previous, next } = useCursorPaginator();
```

and using it like this:

code/4-issue-finder/src/App.5.tsx

```
51     <PaginationControls
52       {...pageInfo}
53       onPrevious={() => previous()}
54       onNext={() => next(endCursor)}
55     />
```

The `usePaginationHook` is a simple implementation to accelerate back navigation and even allows us to enhance the UI to enable the user to navigate back more than one page.

The hook however doesn't allow us to navigate backwards past the visited pages. If you are up for a small challenge I suggest you try to add those features.

Summary

In this chapter we've built the foundation for a type-safe React and Apollo app. In the next chapter we're going to round it out by adding pre-fetching queries, loading comments, and creating mutations.

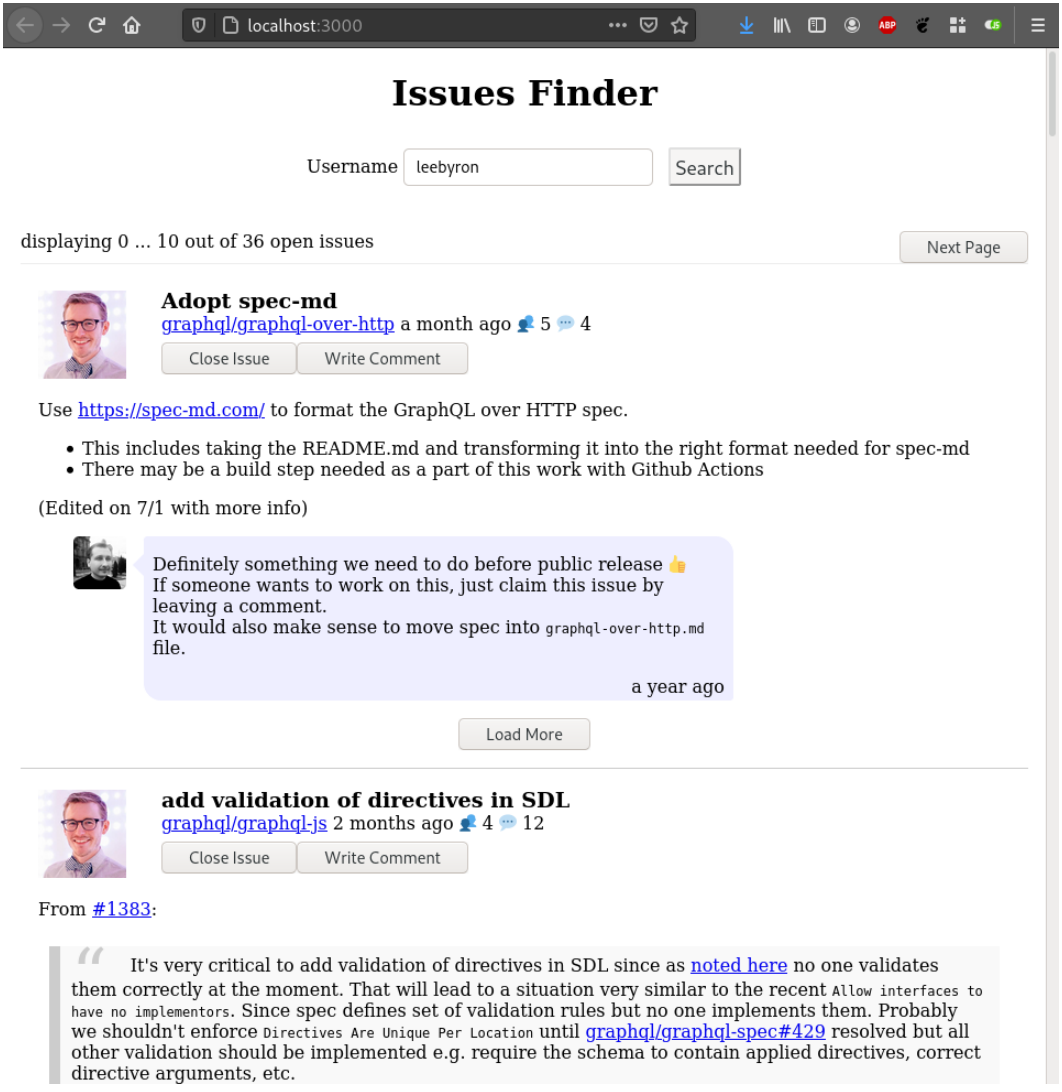
Building a TypeSafe GraphQL React Client App - Part 2

What are we building?

In this chapter, we're continuing the app from the previous chapter.

In this chapter you'll learn:

- Improve the performance of your app
- Mutations and some details on how to use them



IssueFinder

Prefetching Queries

In the last chapter, the `usePaginationHook` is a simple implementation to accelerate back navigation and even allows us to enhance the UI to enable the user to navigate back more than one page.

Previously, we made the navigation to the previous page feel instantaneous. Can we accomplish the same with the next page?

We just learned that Apollo caches all queries we send, so how about ensuring that pagination to the next page loaded as soon as the user clicks on next? - we can do this by prefetching the next page.

All we need to do, is take the `apolloClient` and execute our query with the next cursor before the user clicks on "Next Page". For simplicity we can do this as soon as the component has been rendered. To achieve this, we can write another hook for this job:

code/4-issue-finder/src/hooks/index.ts

```
20 export const useQueryPrefetcher = (query: DocumentNode, options: {}) =>\
21  {
22   const apolloClient = useApolloClient();
23
24   return apolloClient
25     .query({
26       query,
27       ...options,
28     })
29     .catch((error) => ({ error }))
30     .then(console.log);
31 };
```

This hook uses `useApolloClient` to ensure we use the same query that has been passed via `<ApolloProvider/>`.

It's important to use the same client as we want to use the same cache. Then we wrap our query in `useEffect` and make it dependent on the input variables to ensure that we only prefetch once.

When we call `usePrefetchQuery` then it will run as soon as the component renders but will not trigger a re-render.

This is how we can use it in our Issue Component:

code/4-issue-finder/src/App.5.tsx

```
32   useQueryPrefetcher(GetIssuesMinimalWithPaginationDocument, {
33     variables: {
34       login,
35       after: endCursor || null,
36     },
37   });
```

If the user now clicks on "Next Page" it will either cause Apollo client to instantaneously return the cached response or subscribe to a request that is already in-progress reducing the waiting time.

This technique can be used beyond pagination and should be considered to speedup query that will be executed with high likelihood.

Loading Comments

Questions without answers not very useful. So let's try to load the comments as well.

We have several ways we can approach this - each with their own advantages.

I'll go through the following options and we will explore their merits:

1. extend the existing `getIssues` Query
2. create a dedicated query
3. Hybrid Approach

The 3 approaches will be exploring how we can trade of number of round trips, latency, amount of data loaded and the resulting user experience.

As we cannot load all comments at once, a pagination solution will be needed but we will be covering this once we discussed all 3 approaches first.

Extending GetIssues Query

Extending existing queries to fit new requirements, is by far the simplest approach and should come naturally.

All we need to do is add `comments` in our `GetIssuesQuery`:

code/4-issue-finder/src/queries/get-issues.full.tsx

```
51            comments(first: $commentCount) {
52              totalCount
53              pageInfo {
54                endCursor
55                hasNextPage
56              }
57              nodes {
58                id
59                author {
60                  avatarUrl
61                  login
62                }
63                updatedAt
64                bodyHTML
65              }
66            }
```

This now fetches the first 10 comments of each of the first 10 open issues. This change nests the comments inside our issues and can be accessed by drilling into the issue object.

We can build a small component for this and directly embed it into the `IssueCard` component. We could use this data like so:

code/4-issue-finder/src/components/Responses.dumb.tsx

```
1  import React from "react";
2  import moment from "moment";
3  import { useGetCommentsQuery, IssueComment } from "../graphql";
4
5  export interface ResponsesProps {
6    nameWithOwner: string;
7    number: number;
8  }
9
10 const Responses = ({ nameWithOwner, number }: ResponsesProps) => {
```

```
  const [owner, name] = nameWithOwner.split("/");
  const { data } = useGetCommentsQuery({
    variables: {
      owner,
      name,
      number,
    },
  });

  if (!data) return null;

  const nodes = (
    data?.repository?.issue?.comments?.nodes || []
  ) as IssueComment[];

  const responses = nodes
    .map((x) => (
      <div className="Response" key={x!.id}>
        <div className="Info">
          <img src={x?.author?.avatarUrl} alt={x?.author?.login} />
        </div>
        <div className="Message">
          <div dangerouslySetInnerHTML={{ __html: x?.bodyHTML }} />
          <span> {moment(x!.updatedAt).fromNow()}</span>
        </div>
      </div>
    ));

  return (
    <div className="ResponsesContainer">
      <div>{responses}</div>
    </div>
  );
};

export default Responses;
```

First we pull out the comments (if they exist) and then we map each of the `IssueComments` to a React elements.

But once we actually run the page you might notice something: loading the data took longer!

A quick back of the envelope calculation shows:

10 issues with 10 comments ⇒ 100 items! We are downloading up to 10 times more data! This is combinatorics in action and is something you need to consider in your design.

We can see here that GraphQL allows us easily to specify and fetch what we want in one go... But this begs the question: do we need all data immediately?

Dedicated Query

As we just saw, it might be not desirable in all cases to fetch all data in one go. One way of dealing with this is by splitting out comments into a dedicated query.

With Github, one can fetch comments by repository and issue number:

code/4-issue-finder/src/queries/get-comments.tsx

```
1  import gql from "graphql-tag";
2  export default gql`
3    query GetComments(
4      $owner: String!
5      $name: String!
6      $number: Int!
7      $after: String
8      $first: Int = 7
9    ) {
10     repository(owner: $owner, name: $name) {
11       issue(number: $number) {
12         comments(first: $first, after: $after) {
13           totalCount
```

```
14            pageInfo {
15              hasPreviousPage
16              startCursor
17            }
18            nodes {
19              id
20              author {
21                avatarUrl
22                login
23              }
24              updatedAt
25              bodyHTML
26            }
27          }
28        }
29      }
30    }
31  `;
```

In order to integrate this we just need to use the automatically generated `useGetCommentsQuery` inside the response component.

code/4-issue-finder/src/components/Responses.simple.tsx

```
5   export interface ResponsesProps {
6     nameWithOwner: string;
7     number: number;
8   }
9
10  const Responses = ({ nameWithOwner, number }: ResponsesProps) => {
11    const [owner, name] = nameWithOwner.split("/");
12    const { data } = useGetCommentsQuery({
13      variables: {
14        owner,
15        name,
16        number,
```

```
    },
  });
```

In this case we also needed to pass the variables to fetch the comments.

Once this is done, we should see the app perform slightly differently:

The app displays the issues quickly and then follows up for every Issue with up to 10 Comments for each issue - in total up to 100 comments.

This reduces the latency and time until the first Issues are displayed. For the sake of responsiveness, we decided to fetch our data in multiple trips but what else can be done?

One avenue one might explore is explicit or lazy loading of comments. And it turns out Apollo provides a handy pattern for that also - `useLazyQuery`!

`useLazyQuery` is very similar to `useQuery` with one exception: it does not fetch data until you invoke a callback.

code/4-issue-finder/src/components/Responses.lazy.tsx

```
export interface ResponsesProps {
  nameWithOwner: string;
  number: number;
}

const Responses = ({ nameWithOwner, number }: ResponsesProps) => {
  const [owner, name] = nameWithOwner.split("/");
  const [load, { data }] = useGetCommentsLazyQuery({
    variables: {
      owner,
      name,
      number,
    },
  });
```

The first element in the result is the callback that will start our request and the second is the state of our query.

code/4-issue-finder/src/components/Responses.lazy.tsx

```
38   return (
39     <div className="ResponsesContainer">
40       <div>{responses}</div>
41       {data ? null : <button onClick={() => load()}> Load Comments </bu\
42 tton>}
43     </div>
44   );
```

We then just need to add our button that calls the `load` callback. And once the data is loaded we hide the button.

This is a very simple way to ensure that we only load what we need. Furthermore this is completely independent of the `GetIssueQuery` and hence trivial to extend with a pagination solution we've built for our `IssueComponent`.

Hybrid Approach

The idea is to fetch a low amount of comments for each issue and load more if the user clicks on the button. To make this work efficiently we need to skip over comments we already fetched via `GetIssuesFullQuery`.

We only need small changes in the Response component to accomplish this:

code/4-issue-finder/src/components/Responses.hybrid.tsx

```
9  export interface ResponsesProps {
10   nameWithOwner: string;
11   number: number;
12   comments: IssueCommentConnection;
13 }
14
15 const Responses = ({ nameWithOwner, number, comments }: ResponsesProps)\
16  => {
17   const [owner, name] = nameWithOwner.split("/");
18   const [load, { data }] = useGetCommentsLazyQuery({
19     variables: {
```

```
      owner,
      name,
      number,
      after: comments.pageInfo.endCursor,
      first: 7,
    },
  });

  const nodes = ((comments.nodes || []) as IssueComment[]).concat(
    (data?.repository?.issue?.comments?.nodes || []) as IssueComment[]
  );
```

The `ResponseProps` now accept a list of issues which then get merged with the data returned by the lazy query. We can combine this approach easily with our `useCursors` hook to add pagination and can recommend you try adding this feature.

This approach allows us to fine-tune the performance of the app by tweaking how much gets loaded in the first roundtrip and how much in the second.

Mutations - Modifying Existing Data

Searching and reading Issues is nice - and if you are a Github power user you'll probably find many old issues that should have been closed a long time ago. Let's make this possible from our app.

The mutation query could look like this:

code/4-issue-finder/src/mutations/close-issue.tsx

```
import gql from "graphql-tag";
export default gql`
  mutation CloseIssue($issueId: ID!) {
    closeIssue(input: { issueId: $issueId }) {
      issue {
        id
        state
      }
    }
  }
`;
```

The query does what you'd expect, it closes the issue with `$issueId` and then returns `id, state` of that issue.

After running `yarn generate` we can then incorporate our mutation by adding `useCloseIssueMutation` to our `IssueCard` component:

code/4-issue-finder/src/components/IssueCard.tsx

```
  const variables = {
    issueId: id,
  };

  const [owner, name] = repository.nameWithOwner.split("/");

  const [closeIssue, { loading: isClosing }] = useCloseIssueMutation({
    variables,
  });

  const [reopenIssue, { loading: isReopening }] = useReopenIssueMutatio\
n({
    variables,
  });
```

This uses the `useMutationHooks` to acquire callbacks to `closeIssues`, `reopenIssues` and the state of those mutations.

A key thing to understand about these mutation hooks:

They don't fire unless you call the callback functions.

So we need to bind them to something - this case, we'll bind them to `onClick` event handlers.

code/4-issue-finder/src/components/IssueCard.tsx

```
105           <div>
106             {isOpen && (
107               <button disabled={isChangingState} onClick={() => closeIs\
108 sue()}>
109                 Close Issue
110               </button>
111             )}
112
113             {isClosed && (
114               <button disabled={isChangingState} onClick={() => reopenI\
115 ssue()}>
116                 Reopen Issue
117               </button>
118             )}
```

The code block shows we bind our `closeIssue()`, `reopenIssue()` to conditionally rendered buttons. The `isChangingState` is derived from:

code/4-issue-finder/src/components/IssueCard.tsx

```
89   const isOpen = state === IssueState.Open;
90   const isClosed = state === IssueState.Closed;
91   const isChangingState = isClosing || isReopening;
```

Once the button is clicked, it will send a the mutation to Github and transition to a disabled state until the query completes.

This may look very simple but there is a lot going on in the depths of the Apollo client. Here is the question one needs to ask:

> Why does *state* change and components re-render once the query completes?

The data from `IssueCard` comes from our `GetIssues` Query, so how did Apollo know that it needed to re-run this query?

Looking into the network tap will show that besides our mutation, no other queries have been made.

What happens is something else: Apollo maintains a cache but when our Mutation returned a new `Issue`, Apollo picked it up and updated the state field of the issue with the returned id.

Then it re-ran all components that use a hook that is associated with potentially stale data. This may sound pretty smart and I suggest you verify it for yourself by removing the `state` from our Mutation:

```
mutation CloseIssue ($issueId: ID!) {
  closeIssue(input: {issueId: $issueId}) {
    issue {
      id
    }
  }
}
```

If you use this query `state` inside `IssueCard` won't update. This leads us to some important guidelines:

1. Every entity should have an unique identifier called `id`
2. Mutations should return the entities that have been modified
3. **If you perform a mutation make sure you query all fields that may have changed**

> If an endpoint doesn't or can't provide `#1` don't worry, there is another solution. I'll talk about it at the end of this chapter.

Mutations - Creating New Data

We just added a the ability to close issues but our app wouldn't be complete if we couldn't also write a comment.

The approach of adding this functionality is no different than to the `CloseIssue` functionality, but there is a small twist.

So let's add the functionality and see how it goes:

code/4-issue-finder/src/mutations/write-comment.tsx

```
import gql from "graphql-tag";
export default gql`
  mutation AddComment($issueId: ID!, $body: String!) {
    addComment(input: { subjectId: $issueId, body: $body }) {
      commentEdge {
        cursor
        node {
          id
          bodyHTML
          updatedAt
          author {
            login
            avatarUrl
          }
        }
      }
    }
  }
`;
```

The snippet above does what you'd expect: it adds a new comment with the content `$body` to the issue with the `$issueId`. We then get the `cursor` and the issue `id`, `body` and `author` but more on that later.

Next we need a form component to manage writing and submission:

code/4-issue-finder/src/components/ActionForm.tsx

```tsx
import React, { useRef } from "react";

export interface ActionFormProps {
  show: boolean;
  isLoading: boolean;
  onSubmit: (text: string) => void;
}

const ActionForm = ({ show, onSubmit, isLoading }: ActionFormProps) => {
  const inputRef = useRef<HTMLTextAreaElement>(null);

  if (!show) return null;

  const submit = () => {
    if (inputRef?.current?.value) {
      onSubmit(inputRef.current.value);
    }
  };

  return (
    <div className="ActionForm">
      <textarea disabled={isLoading} ref={inputRef} />
      <br />
      <button disabled={isLoading} onClick={submit}>
        Submit
      </button>
    </div>
  );
};

export default ActionForm;
```

This is a rudimentary component that shows a `textarea` and a `button`. It uses an `inputRef` to extract the value of the text area on submission.

We then can use this and our mutation in the `IssueCard` to implement our desired feature:

code/4-issue-finder/src/components/IssueCard.tsx

```
78   const onSubmit = (body: string) => {
79     addComment({
80       variables: {
81         issueId: id,
82         body,
83       },
84     });
85   };
```

This is very similar as for `useCloseCommentMutation` but the devil is in the details.

Because we can't know all variables upfront, we didn't pass any variables to the mutation hook and instead pass them in `onSubmit` into the `addComment` callback.

Refetching Queries

We are almost done. But we have a problem: if the user submits a comment it won't appear in our app!

This is where Apollo expects us to fiddle with it's cache. One approach is to just tell apollo what queries need refetching by providing `refetchQueries` to `useAddCommentMutation`:

```
useAddCommentMutation({
  refetchQueries: [{
    query: GetIssuesDocument,
    variables: currentSearchQueryVariables
  }]
})
```

This tells Apollo to **refetch the specified list of queries with the provided variables**.

What will happen is that it will refetch all issues as soon as the mutation completes.

While this is convenient, it might be excessive and we can do something more refined: manually add the new entry into Apollo's cache.

Manually Updating the Apollo Cache

This will be performed in 3 steps:

- reading data from cache using `readQuery`
- merging new data with the cached data
- writing data to cache using `writeQuery`

This is how to do it:

code/4-issue-finder/src/components/IssueCard.tsx

```
50   const [addComment, { loading: isAddingComment }] = useAddCommentMutat\
51 ion({
52     update: (cache, result) => {
53       const data = cache.readQuery<GetIssuesFullQuery>({
54         query: GetIssuesFullDocument,
55         variables: queryVariables,
56       });
57
58       if (!data?.user?.issues?.nodes?.length) {
59         return;
60       }
61
62       const index = (data?.user?.issues?.nodes || []).findIndex(
63         (x) => x?.id === id
64       );
65       const updated = update(
66         data as {},
67         `user.issues.nodes.${index}.comments.nodes`,
68         (comments) => [result?.data?.addComment?.commentEdge?.node, ...\
```

```
comments]
      );

      cache.writeQuery({
        query: GetIssuesFullDocument,
        variables: queryVariables,
        data: updated,
      });
    },
  });
```

Apollo will call this `update` when the mutation was successful.

Our `update` method will receive the `cache` and the `result`. We then read out the cache using `cache.readQuery` which is very similar to `client.query` or `useQuery` but only returns the data in the cache without ever making a network request.

In the second part we try to find our issue in the list of issues found `user.issues.nodes`. The nesting is the result of our query `GetIssuesQuery`. To quickly update the key we'll use `lodash.update` and splice in our comment into the list of comments of our issue.

Lastly we write our updatedData using `cache.writeQuery`. WriteQuery replaces the data in apollos cache and ensures all components that all subscribers - in our case all components that subscribed with `useQuery` - receive the new data.

Summary

GraphQL schema-driven approach allows us to create type-safe React apps and paired with `graphql-codegen` we can be confident that our types are always in-sync. We explored in this chapter how to build an app from scratch, learned how to tackle the challenges of datafetching and caching with graphql. Remember:

- always run `graphql-codegen` to ensure your queries are up-to-date.
- to always query the `id` of entities of your queries to ensure apollo caches your queries
- prefetch and use pagination to keep things responsive

- after a mutation you have to either refetch queries or update the cache manually to update the stale data

What's Next?

We now saw a fair amount of the Client side world of GraphQL and we could go even deeper. For instance Apollo provides great facilities for state management, automatic re-fetching and via apollo-link we can even query traditional REST APIs using GraphQL.

However, I think it's time to see the other side of the story and the next chapter will cover the server-side aspects of GraphQL.

FAQ

What happens if an endpoint does not provide an unique Id as a field?

```
mutation CloseIssue ($issueId: ID!) {
  closeIssue(input: {issueId: $issueId}) {
    issue @key(fields: "id") {
      id
    }
  }
}
```

and in `GetIssuesQuery` it would look like:

```
query GetIssues() {
  repository(...) {
    Issues {
      nodes @key(fields: "id") {

      }
    }
  }
}
```

The unique key can also consist of multiple fields. For illustration purposes image this set of types:

```
type Nested {
 key3: String
}

type Object {
  key1: String
  key2: Int

  nested: Nested
}

type Query {
  objects: Object
}
```

We can then use `@key` this way to compute a unique key using key1 key2 key3:

```
query GetObjects {
  objects @key(fields: "key1 key2 nested { key3 } {
    ...
  }
}
```

Summary

This chapter was built on top of the previous chapter, where you create a basic React application with GraphQL using Apollo client. In this successive chapter, the application was bootstrapped with `create-react-app` and `graphql-codegen`, allowing you to automatically create Apollo React components from your GraphQL schema and improve the performance of your application. Finally, GraphQL mutations and their intimacies were discussed.

After working with existing GraphQL APIs, the next chapter will teach how to create a GraphQL server by yourself.

Your First GraphQL Server with Apollo Server

In previous chapters, you've been using GraphQL client-side, and you have learned that the shape of responses is very flexible and fully determined by the documents you send to the server.

Unlike frameworks to build REST APIs, GraphQL server frameworks are designed from the ground up to efficiently support this flexibility. GraphQL servers are very opinionated and force you to write your request handlers – called **resolvers** – in a very specific way.

In this chapter, you'll learn to:

- Set up a small GraphQL endpoint with Express and Apollo
- How to structure schemas
- Creating resolvers
- Handling documents by creating a user profile service

Getting started

To get you started a bare bone Node.js application in TypeScript can be found in the directory `code/5-apollo-server/`. Using a minimal amount of packages and some configuration, this provides you with the fundamentals to create a GraphQL server on top. In the directory you need to download the necessary packages with npm (or yarn, if you prefer):

```
npm i
```

Once npm has finished the installation, you'll need to add the libaries `express` and `apollo-server-express` that make it possible to create a GraphQL server:

```
npm i express apollo-server-express
```

After installing, you can proceed to the next section of this chapter to start building your first GraphQL server.

Schema

Let's start with the schema. We'll create a new file called `schema.ts` in the `src` directory, and add the following code:

code/5-apollo-server/src/schema.1.ts

```
import { gql } from 'apollo-server-express';

export default gql`
  type User {
    id: ID!
    firstName: String!
    lastName: String!
    age: Float
    email: String
  }

  type Query {
    users: [User]
  }
`;
```

We create this schema using the `gql` function that we covered in *Chapter 2*. The function parses the string between the backticks and returns the schema as an AST.

> You don't have to write your GraphQL schema using a template literal. Alternatively, you can create a `.graphql` file and add it to your build process. This requires additional configuration using `babel` plugins or Webpack.

In the code above, we have a type `User` that consists of:

- a required integer field `id`,
- required string fields `firstName` and `lastName`,
- an optional float field `age`, and
- an optional string `email`.

Types of the fields are a guarantee for consumers and resolvers alike: the GraphQL engine will rather throw an error and fail than break this promise.

We also have the `Query` operation type that defines which queries your server will accept and resolve. For now, we only added `getUsers` that will return an array of `Users`.

> In addition to `Query`, GraphQL schemas can also have `Mutation` and `Subscription` operation types. These are the only three operation types that are generally accepted by GraphQL servers.

Now that we have a schema, we can continue by adding boilerplate code for your GraphQL server.

The Obligatory Boilerplate

Here is the minimum boilerplate you need to get a server up and running. You should add this code to `index.ts` instead of its current contents:

code/5-apollo-server/src/index.1.ts

```
import express from 'express';
import { ApolloServer } from 'apollo-server-express';

import typeDefs from './schema';

const server = new ApolloServer({
  typeDefs,
});

const app = express();
server.applyMiddleware({
  app,
  cors: true,
});

const PORT = 4000;

app.listen(PORT, () => {
  console.log(
    `GraphQL endpoint and playground accessible at http://localhost:${P\
ORT}${server.graphqlPath}`,
  );
});
```

Here we use a "batteries included" Apollo Server for the API framework of our choice. In our case it's `apollo-server-express` for `express`, but in the comments below the imported modules, there are plenty of alternatives listed to choose from.

We then need to import `typeDefs`, which is your **schema**, and our `resolvers`. These are the two bits that Apollo Server needs to create the GraphQL server, and we'll dive into how to write these next.

The code block below instantiates the Express server and extends it with the GraphQL server functionality:

code/5-apollo-server/src/index.1.ts

```
10 const app = express();
11 server.applyMiddleware({
12   app,
13   cors: true,
14 });
```

The GraphQL server is then linked to the `/graphql` endpoint, for security reasons `Cross-Origin Resource Sharing (CORS)` is enabled.

> *CORS* is a security feature that can enable or disable browser access to your GraphQL server from domains other than yours.

Finally, the method `app.listen` starts your server at the provided port `4000`. This makes a GraphQL endpoint available and creates a GraphQL playground like the one you used in *Chapter 1*. However, before you can actually start the server, you'll need to create resolvers that get your data, and we'll cover this in the upcoming sections.

Mocking the Data

Before we start writing resolvers, let's use mock data to test the schema. You can do this as soon as you've created a schema – just add `mocks: true` to Apollo Server options:

code/5-apollo-server/src/index.2.ts

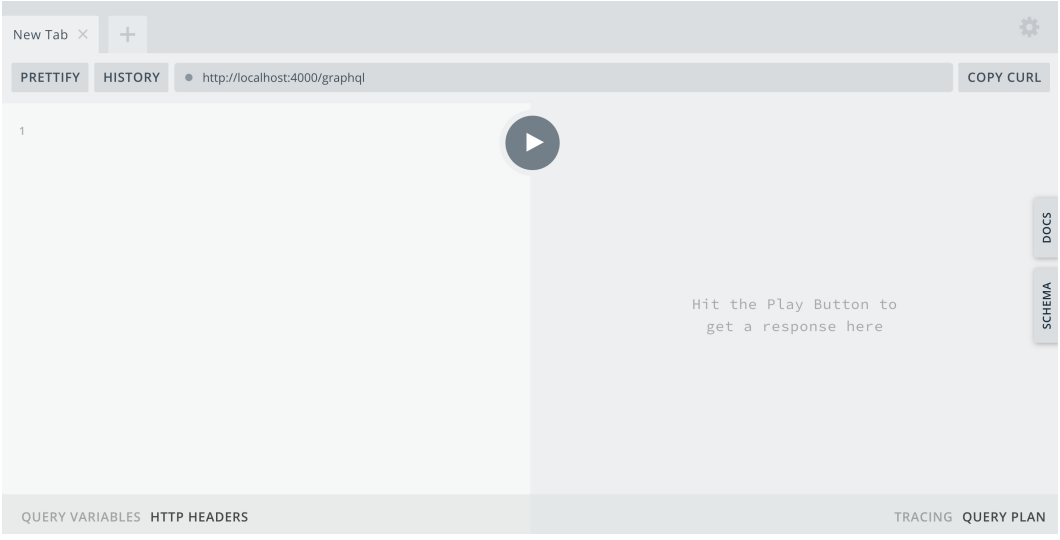
```
6 const server = new ApolloServer({
7   typeDefs,
8   mocks: true,
9 });
```

This will generate dummy data, enabling you to prototype your UI and incrementally develop your server logic.

In the `package.json` file, the `start` script runs `ts-node` with `nodemon` to compile and execute the file `src/index.ts`. This allows you to start the GraphQL server by running:

```
npm start
```

After starting the GraphQL server, you can inspect it in the GraphQL playground available at `http://localhost:4000/graphql`. This playground looks similar to this:

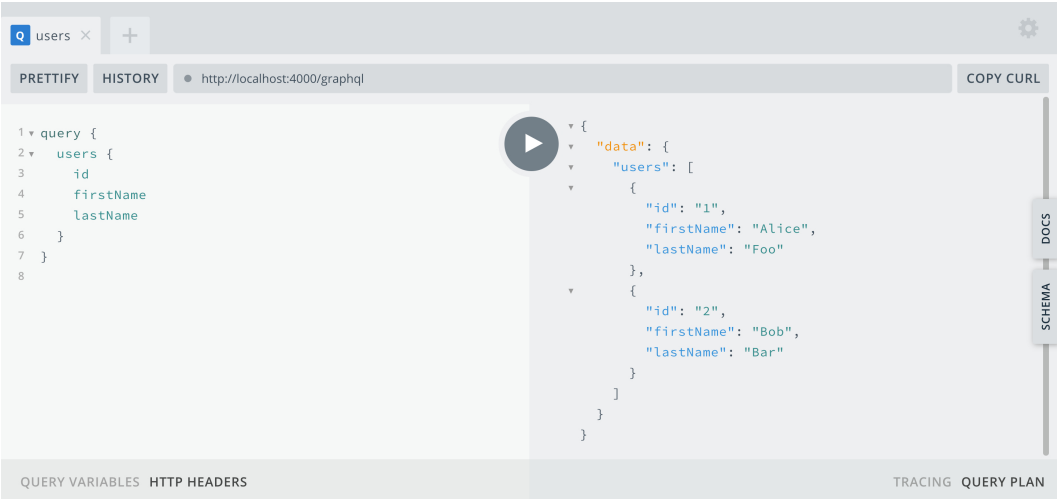


The GraphQL playground

Once in the playground, try sending the following document on the left-hand side of the screen:

```
query {
  users {
    id
    firstName
    lastName
  }
}
```

The server will respond with an array that contains objects with mock values for `id`, `firstName` and `lastName`, as you can see on the screenshot of the playground below.



Showing results in the GraphQL playground

Returning mock data is just the start, as you can also use Apollo Server to configure the mock data.

Configuring Mocks

In addition to generating mocks for you, Apollo Server also lets you configure mock values by passing an object to the `mocks` field. The `mocks` object is a map from type to value, for instance:

code/5-apollo-server/src/index.3.ts

```
6 const mocks = {
7   Int: () => 42,
8   Float: () => Math.random(),
9 };
```

The server will now always return `42` for every `Int` field, which in our case is the field `id`, and a random value for fields of type `Float`, such as `age`.

You can even configure your types to have a specific shape:

code/5-apollo-server/src/index.4.ts

```
const mocks = {
  User: () => ({
    id: 1,
    firstName: 'foo',
    lastName: 'bar',
  }),
};
```

When you send a document with the query you've just used, the GraphQL server will now return a specific shape, as this mock constant is acting as a resolver. A popular library to generate more dynamic dummy data is `casual`, which provides a plethora of generators.

Let's continue and see how to resolve actual data.

Resolvers

Resolvers contain logic to retrieve your data in the structure that you've defined in the schema. The mock function you created in the previous part is probably the simplest resolver that can satisfy our contract, as it just returns hard-coded data. However, a resolver could just as well return a promise with data that came from a database.

> Resolver functions can be either synchronous or asynchronous.

Let's create resolvers in a new file called `src/resolvers.ts`, where you need to define a function that will get the list of users. This function, `getUsers`, will return the list of users from a simple `Map` object that we import from `./database` as `db` , so we're not using an actual database yet. To retrieve all entries of a `Map` object, you can turn the values of the object into an array:

code/5-apollo-server/src/resolvers.1.ts

```
import db, { User } from './database';

function getUsers(): Array<User> {
  return Array.from(db.users.values());
}

const resolvers = {
  Query: {
    users: getUsers,
  },
};

export default resolvers;
```

> Note: You defined a *GraphQL* type in the schema above, but here you're defining a *TypeScript* type. These are two different things, but there are some similarities in how types and values are defined. Later on, you'll learn how to automatically generate one from the other, but for now, you can define it manually as shown above.

The result of this function will be of shape `User` that you've just defined, which has the same fields as your `User` type in the GraphQL schema. Let's start by returning a fixed array of objects:

The next thing you need to do is connect the resolver function above to the schema that you've defined before. To do so, resolvers need to be mapped to operation types in a *resolver-map*. Here's what it looks like:

code/5-apollo-server/src/resolvers.1.ts

```
const resolvers = {
  Query: {
    users: getUsers,
  },
};
```

The resolver-map is a mapping between **types** defined in the schema and **resolvers** that retrieve the data. Apollo Server will execute `resolvers.Query.getUser` to resolve data for any query with the following shape:

```
query {
  users {
    # ...fields
  }
}
```

The result of `resolvers.Query.getUser` will then be validated against the schema. Before returning a response to the client, fields that weren't requested will be stripped off.

To demonstrate this, let's import resolvers into the file `src/index.ts` and add them to Apollo Server's options object:

code/5-apollo-server/src/index.5.ts

```
import typeDefs from './schema.5';
import resolvers from './resolvers.6';

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

Now, if you open the GraphQL playground again and send the document with a query to retrieve users, the data will be returned as defined in `src/resolvers.ts`.

Not only can **resolvers** be defined for operation types in your schema, but they can also be defined on the field level. The GraphQL engine inspects the query embedded in a request's document, and invokes all relevant resolvers. It will traverse the result object (of `getUsers`), call every resolver function, wait until every promise resolves (or rejects), and validate the final result against the provided schema.

This process is lazy, and it will only traverse fields that are important to the request's document. To take full advantage of this laziness, you need to split resolvers into small chunks.

Let's extend the initial schema to demonstrate this:

code/5-apollo-server/src/schema.2.ts

```
4   type User {
5     id: ID!
6     firstName: String!
7     lastName: String!
8     name: String! # This is the computed full name of a user
9     age: Float
10    email: String
11  }
```

This adds a required field `name` to the schema. However, if you write a resolver for `name`, it doesn't need to be retrieved directly by the resolver function `getUsers`. Instead, it can be defined in the resolver-map in `src/resolvers.ts` along with a function to compute this field:

code/5-apollo-server/src/resolvers.2.ts

```
7   function computeName(user: User): string {
8     console.log(`Computing name for`, user);
9     return `${user.firstName} ${user.lastName}`;
10  }
11
12  const resolvers = {
13    User: {
14      name: computeName,
15    },
```

```
  Query: {
    users: getUsers,
  },
};
```

Apollo Server will now augment each item of the array returned by `getUsers` with a value for `name`. The `name` field will be computed by invoking the function `computeName`.

Here `computeName` receives a `User` object that doesn't have a field and value for `name`. It only calls the function if `name` is defined in the query that was passed in the document to the GraphQL server. You can verify this by looking at the output of the `console.log` method in `computeName`, between the response of this query that asks for the `email` of every user:

```
query {
  users {
    email
  }
}
```

Here's a query that requests `name` for every user:

```
query {
  users {
    name
  }
}
```

This latter query should invoke `computeName` for every user and trigger the `console.log` method. To verify this, head over to the playground at `https://localhost:4000/graphql`, or send a document with each of these queries using cURL.

In your console that runs the GraphQL server, the following output will be logged:

```
Computing name for User {
  id: 2,
  firstName: 'Bob',
  lastName: 'Bar',
  age: 27,
  email: null
}
```

Besides computing fields in your schema using the resolvers, you can also chain resolvers as you'll learn in the next section.

Chaining Resolvers

Imagine that you need to add a new type called `Post` that has a relation with the `User` type and must be added to the operation type `Query`. After adding this new type and its relations, your GraphQL schema would look like this:

code/5-apollo-server/src/schema.3.ts

```
1  import { gql } from 'apollo-server-express';
2
3  export default gql`
4    type Post {
5      id: ID!
6      title: String!
7      body: String!
8    }
9
10   type User {
11     id: ID!
12     firstName: String!
13     lastName: String!
14     name: String! # This is the computed full name of a user
15     age: Float
16     email: String
17     posts: [Post]
```

```
  }

  type Query {
    users: [User]
    posts: [Post]
  }
`;
```

This code introduces queries for posts, and posts are related to users. However, before you're actually able to query all posts or posts by a specific user, you need to create a resolver for `Post`. Posts can be retrieved from the `Map` database as well:

code/5-apollo-server/src/resolvers.3.ts

```
import db, { User, Post } from './database';

function getPosts(): Array<Post> {
  return Array.from(db.posts.values());
}
```

> You don't have to add the field `posts` to the `User` type in TypeScript because the types that are defined here are only used for resolvers and not GraphQL operations.

Before you're able to query the posts, you'll need to add them to the resolver-map below the `users` query:

code/5-apollo-server/src/resolvers.3.ts

```
  Query: {
    users: getUsers,
    posts: getPosts,
  },
```

You can now query all posts by sending a document with the following query:

```
query {
  posts {
    title
    body
  }
}
```

Remember how we retrieved the `name` field in a lazy manner? Similarly, we can make GraphQL resolve posts for every user through the resolver-map. Let's create a new function that takes a user object as an argument and returns all posts for the user. We will retrieve all posts from the `Map` database, and then filter results by user ID:

code/5-apollo-server/src/resolvers.4.ts

```
16 function getPostsByUser({ id }: User): Array<Post> {
17   const posts = Array.from(db.posts.values());
18
19   return posts.filter(({ authorId }) => authorId === id);
20 }
21
22 const resolvers = {
23   User: {
24     name: computeName,
25     posts: getPostsByUser,
26   },
27   Query: {
28     users: getUsers,
29     posts: getPosts,
30   },
31 };
```

GraphQL now only resolves posts if needed, and takes care of merging posts into users for you. This means you can send queries to get `users` with fields `title` and `body` for their `posts` and the (computed) `name` for every user:

```
query {
  users {
    name
    posts {
      title
      body
    }
  }
}
```

Given that resolving data is lazy, we can have circular references in our types and also relate users to posts the other way around. This would require you to add a relationship for `Post` with `User` in the GraphQL schema:

code/5-apollo-server/src/schema.4.ts

```
4   type Post {
5     id: ID!
6     title: String!
7     body: String!
8     author: User
9   }
```

Now, create a resolver function to get the user for a post, and add it to the resolver-map for `Post`. As the database is a `Map`, a specific entry for `users` can be retrieved by passing an ID to the `get` method:

code/5-apollo-server/src/resolvers.5.ts

```
22  function getAuthorForPost({ authorId }: Post): User | undefined {
23    return db.users.get(authorId);
24  }
25
26  const resolvers = {
27    User: {
28      name: computeName,
29      posts: getPostsByUser,
```

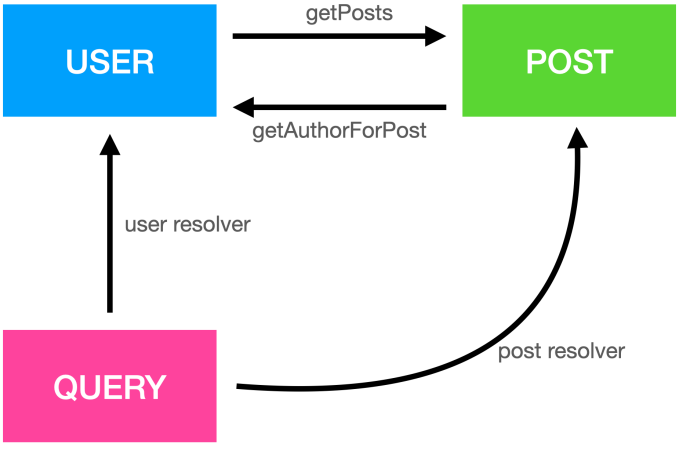
```
  },
  Post: {
    author: getAuthorForPost,
  },
  Query: {
    users: getUsers,
    posts: getPosts,
  },
};
```

This allows you to retrieve the `name` of an author for every post by sending this query in a document:

```
query {
  posts {
    title
    body
    author {
      name
    }
  }
}
```

GraphQL can resolve these fields because when a field has a resolver function, the parent object is always sent to the resolver function as an argument. This is due to the conceptual way of thinking in graphs that is applied in GraphQL.

Thinking in graphs is at the core of GraphQL, and can best be explained by thinking of types as *nodes* and resolvers as *edges*. That way you would get a graph like this:



GraphQL Nodes and Edges

This is why it's called GraphQL. We have nodes and edges, and the GraphQL query language helps transform this graph to a response.

The GraphQL engine leverages resolvers to traverse the graph and passes the previous "node" to the resolver to retrieve the next "node". Consider this query as an example:

```
query {
  users {
    name
    posts {
      title
      body
    }
  }
}
```

The GraphQL engine calls the `getPostsByUser` resolver function for every `user` because the resolver for `posts` is placed in the `User` type in your resolver-map.

You can describe this process as follows:

1. To resolve `user`, call `getUsers`
2. `getUsers` returns an array of type `User`
3. All resolvers for requested fields in the resolver-map for `User` will be checked:
   - To get `posts`, the `getPostsByUser` resolver will be called with the entire object that is currently being resolved
4. `getPostsForUsers` returns an array of type `Post`
5. This process is repeated for all posts

GraphQL accomplished something interesting here, as you started with two arrays in memory: `users` and `posts`. The response of your query, however, is an array of `users` where each user entry contains an array of `posts`.

You might just as well say: GraphQL de-normalized our data!

Passing Arguments

So far, the queries in your schema take no arguments: you're requesting either all users or all posts. In our example, the query for `User` returns all available users, and can't return a selection based on a specific `id` or using an offset for pagination.

To make it possible to return a specific user based on a value for `id`, a new query must be added to the schema that will return a single instance of `User`.

code/5-apollo-server/src/schema.5.ts

```
21  type Query {
22    users: [User]
23    user(id: Int): User
24    posts: [Post]
25  }
```

> The type of `id` defined in the `User` type differs from type of `id` as an input value. This is because the scalar type `ID` always translates to a string.

The query `user` will take a value for `id`, which is a `String`. Based on that string the query will return a user, something that requires you to create a resolver. This resolver is similar to the resolver that gets the author for a post, as you're also requesting a specific user from the `Map` database:

code/5-apollo-server/src/resolvers.6.ts

```
11 type UserArguments = {
12   id: number;
13 };
14
15 function getUser(
16   _: any,
17   { id }: UserArguments,
18   context: any,
19 ): User | undefined {
20   return db.users.get(id);
21 }
```

In the resolvers file, we created a new resolver function that takes three objects as arguments: the parent object, the arguments object, and the context. The parent object can be used when you're querying a field for a type, such as `User`, and is not needed in this case. The context object also isn't used (yet), and the arguments object holds the `id` that was passed to this query.

As you'll already know, this resolver must be added to the resolver-map:

code/5-apollo-server/src/resolvers.6.ts

```
46   Query: {
47     users: getUsers,
48     user: getUser,
49     posts: getPosts,
50   },
```

You can now pass this query from the GraphQL playground at http://localhost:4000/graphql[27] to get both `id` and `name` for the user with `id` "1":

[27] http://localhost:4000/graphql

```
query {
  user(id: 1) {
    id
    name
  }
}
```

This works fine when you're only passing one query, but we can optimize further by changing this `query` to a named query. This is especially important when you're sending queries from a web application, as the names can be used for caching:

```
query GetUser($id: Int) {
  user(id: $id) {
    id
    name
  }
}
```

The value for `$id` can be added in the bottom left side of the playground, where you can pass along this JSON file in the "Query variables" tab:

```
{
  "id": 1
}
```

Using Query Variables

When your queries grow, this is a scalable way of using named queries both in the playground and from your GraphQL client.

Summary

In this chapter, you have learned how to set up a GraphQL server with Express, Apollo and TypeScript. Using mock data and a simple `Map` database, you have created a schema that is used by resolvers to get data for responses to your queries.

Thanks to the async nature of resolvers, GraphQL can also be used to retrieve data from databases and other APIs, and we'll talk about databases in the next chapter. There are even packages to help you get data from different sources more conveniently using just a single library: Dataloader. We will cover the interplay of GraphQL with databases, as well as Dataloader and optimizations, in dedicated chapters, as it's a huge topic!

Using GraphQL Servers with A Database

In the previous chapter, you have learned how to set up a GraphQL server with Apollo and retrieve data from that server using queries. For a deeper dive into GraphQL servers, let's connect an actual database to a server and mutate the data in that database with GraphQL mutations. Also, more advanced queries will be used for instance to handle pagination.

This chapter will cover: - Using GraphQL with a database - Queries with pagination - Writing mutation resolvers - Handling errors

To explore these topics in this chapter, we will create a GraphQL server as the backend for a music application. Using queries and mutations, you will be able to view tracks, add playlists, and save tracks to playlists that you've created.

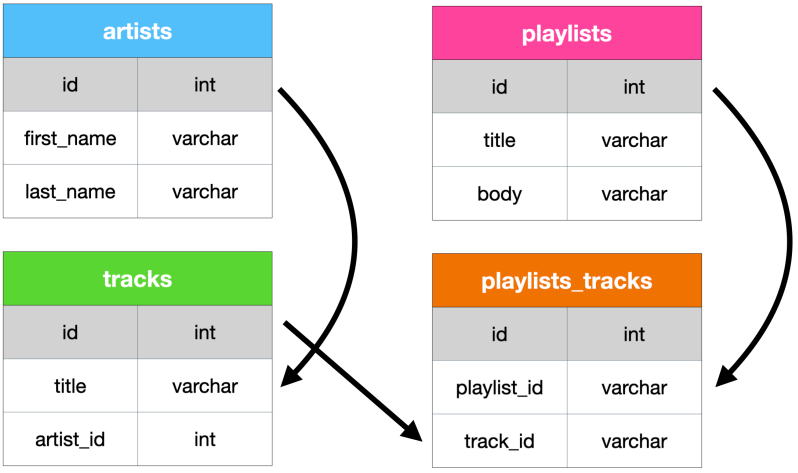
Getting Started

The code examples in this chapter use the same structure as in the preceding chapter and is also using `express` and `apollo-server-express` to construct a GraphQL server. The code can be found in the directory `code/5-apollo-server/` and can be set up by running the following command with npm (or yarn, if you prefer):

```
npm i
```

After the command has finished running, the installation is complete.

Before getting into detail about connecting the GraphQL server to the database, let's have a look at the database design for this server that you'll be building. The database consists of tracks that have a one-to-one relationship with artists and playlists containing these tracks, as shown in the diagram below.



The database schema

This database design is based on a relational database, like SQL. By understanding this design, you have a solid base to learn how to use GraphQL with a database in the next section.

Using GraphQL with a Database

The GraphQL server you've built in the previous chapter used a simple `Map` database that is less advanced than a SQL-based database. In this chapter, we will use an SQLite database called `sqlite3` that is based on the same principles as any other SQL database like MySQL or PostgreSQL.

> This chapter uses a SQL-based database, but the same principles can be applied to other types of databases.

Connecting with a database in a GraphQL server is done from the context, which is an object that is shared between all resolvers. Any data that initiates in the context can be used from every resolver, making it the perfect location to connect to a

database. To reduce the amount of boilerplate that you need to write to query or mutate a database, we will use a low-level SQL query builder called Knex.js.

The logic to start and populate the SQLite database with data is available in the file `src/database.ts` that is included in the starter project. This file exports a function called `startDatabase` that returns the database object `knex` used to query and mutate data. To connect to the database, we need a context object, and we create it in the file `src/index.1.ts` along with the GraphQL server:

code/6-databases/src/index.1.ts

```
6  import startDatabase from './database';
7
8  const server = new ApolloServer({
9    typeDefs,
10   resolvers,
11   context: async () => {
12     const knex = await startDatabase();
13
14     return { knex };
15   },
16 });
```

The `context` field in the options object passed to Apollo Server contains an asynchronous callback. The callback awaits the initialization of the database and adds the database object to the context. Resolvers for queries defined in the file `resolvers.1.ts` receive the context object as the third argument:

code/6-databases/src/resolvers.1.ts

```
9  async function getTracks(_: any, {}, { knex }: Context): Promise<any> {
10   if (!knex) throw new Error('Not connected to the database');
11
12   return await knex('tracks').select();
13 }
```

The first parameter the resolver receives is the parent object – as seen in the previous chapter, this is useful when working with partial resolvers for fields of a type. The

second argument is the object containing the arguments that are passed along with the operation. The third and only typed argument is the context.

If you'd translate the function that communicates with the database to SQL, it will be:

```
SELECT * FROM `tracks`;
```

Every time you'll send a document to query the tracks with, this SQL query will be executed on the database.

This query isn't that complicated, a more complex query can be found if you take a look at the function that returns the tracks for a specific playlist:

code/6-databases/src/resolvers.1.ts

```
42 async function getTracksByPlaylist(
43   { id }: any,
44   {}: any,
45   { knex }: Context,
46 ): Promise<any> {
47   return await knex('playlists_tracks')
48     .where('playlist_id', id)
49     .join('tracks', 'playlists_tracks.track_id', 'tracks.id')
50     .select('tracks.id', 'tracks.title');
51 }
```

This function is called recursively when tracks for a playlist are requested, as you can see from the resolver map. So every time a track for a playlist is requested, the linking table `playlists_tracks` and the `tracks` table are queried.

```
SELECT * FROM `playlists`;
SELECT `playlists_tracks.playlist_id`, `playlists_tracks.track_id`, `tr\
acks.id`, `tracks.title` from `playlists_tracks` INNER JOIN `tracks` ON\
 `playlists_tracks.track_id` = `tracks.id`;
```

> Knex.js uses an `INNER JOIN` by default but you can also choose other join methods like `leftJoin`, `rightJoin`, or left/right outer joins.

In the next section, you'll add more advanced queries to handle pagination on the server.

Queries with pagination

Pagination is a crucial part of every API and allows you to limit the amount of data returned or break up data on different pages. It delivers optimizations on both the server-side by limiting the load on the database, and on the client-side by minimizing the amount of data that must be loaded into the browser.

In GraphQL there are two approaches to handling pagination, which are offset and cursor pagination.

Offset pagination

Generally speaking, offset pagination is the most used form of pagination that is used by APIs. It's also the common approach to handle pagination for REST APIs, and can be easily applied to SQL queries like this:

```
SELECT * FROM `tracks` LIMIT 50 OFFSET 0;
```

The SQL query above will select the first 50 tracks from the database, starting from the first record for `tracks`. Changing the page number will result in a different value for `OFFSET` which is equal to the number of results per page times the page number.

Applying offset pagination to the GraphQL server requires you to add the page number and the number of results per page to your GraphQL schema. After which the schema looks like the following:

code/6-databases/src/schema.1.ts

```
24   type Query {
25     tracks(limit: Int, page: Int): [Track]
26     playlist(id: ID!): Playlist
27   }
```

This change adds the arguments `limit` and `page` to the query to retrieve the tracks, which you then can use in the resolver to change the function that queries the tracks on the database:

code/6-databases/src/resolvers.1.ts

```
9  type Pagination = {
10   limit?: number;
11   page?: number;
12 };
13
14 async function getTracks(
15   _: any,
16   { limit, page = 1 }: Pagination,
17   { knex }: Context,
18 ): Promise<any> {
19   if (!knex) throw new Error('Not connected to the database');
20
21   if (page && limit) {
22     const offset = page - 1 * limit;
23     return await knex('tracks').select().limit(limit).offset(offset);
24   }
25
26   return await knex('tracks').select();
27 }
```

When there's a value for `limit` and `page`, the offset will be calculated that determines which records must be pulled from the database. From the GraphQL playground you can use add values for these arguments to retrieve a limited amount of results when querying for `tracks`:

```
query {
  tracks(limit: 1, page: 2) {
    title
  }
}
```

However, offset pagination has some limitations as it assumes you always want to present your data in pages.

Cursor pagination

Another approach to "paginate" in GraphQL is cursor pagination, which is focused on giving you the results after a specific result - the cursor.

```
query {
  tracks(first: 10, after: 1) {
    title
  }
}
```

The cursor is usually defined by its unique identifier with the parameter `after`, forcing you to make it possible to retrieve the next results from your database by either using a numerical identifier or by for example creation date.

Cursor pagination is often used to create a connection with Relay, which is a client-side implementation of GraphQL created by Facebook. Using GraphQL with Relay is opinionated and requires you to write your resolvers in a way that they return the connection handles for Relay.

Writing Mutation Resolvers

You've already seen that the context object passed to a resolver contains the database object and can be used to access the database. A resolver for a mutation is very similar to the resolvers that you've already seen, and it receives the same parameters.

To create a mutation resolver, you first need to add a mutation to the schema using the `Mutation` type:

code/6-databases/src/schema.2.ts

```
24   type Query {
25     tracks(limit: Int, page: Int): [Track]
26     playlist(id: ID!): Playlist
27   }
```

This mutation takes several arguments, which are all fields available on the `Playlist` type that this mutation returns. Before you can send any document containing a mutation to add a playlist, you should add the `addPlaylist` mutation to resolvers and the resolver-map. The resolver to add a playlist to the database is created in `src/resolvers.ts`, gets the passed arguments from the arguments object, and uses the database object from the context to insert the playlist into the database.

code/6-databases/src/resolvers.3.ts

```
67 async function addPlaylist(
68   _: any,
69   { title, body }: any,
70   { knex }: Context,
71 ): Promise<any> {
72   if (!knex) throw new Error('Not connected to the database');
73
74   const [id] = await knex('playlists').insert({ title, body });
75
76   return {
77     id,
78     title,
79     body,
80   };
81 }
```

The code above shows how to create a resolver for the `addPlaylist` mutation: it uses the database object to insert `title` and `body` into the table `playlists`. The `insert` method from Knex.js returns the last inserted ID (or IDs), which can be used to return the inserted data to the user.

To make this mutation usable in a document, you need to add it to the resolver-map first:

code/6-databases/src/resolvers.3.ts

```
94   Mutation: {
95     addPlaylist,
96   },
```

You can also send this mutation in a document from the GraphQL playground at http://localhost:4000/graphql[28]:

```
mutation {
  addPlaylist(title: "My first playlist", body: "Lorem ipsum...") {
    title
    body
  }
}
```

This would look like the following, seen from the GraphQL playground:

The GraphQL playground with a mutation

[28] http://localhost:4000/graphql

As the number of fields used as input for this mutation grows, it makes sense to create a separate `input` type for the mutation. When you use an `input` type, you no longer need to type the values that you're mutating one by one. Let's add an `input` type to the schema and link it to the mutation:

code/6-databases/src/schema.3.ts

```
29   input AddPlaylistInput {
30     title: String!
31     body: String
32   }
33
34   type Mutation {
35     addPlaylist(input: AddPlaylistInput!): Playlist
36   }
```

The `input` type for the `addPlaylist` mutation contains fields `title` and `body` from the `Playlist` type, of which only `title` is required. The mutation itself requires the user to send an `input` object of type `AddPlaylistInput` along with the document containing this mutation.

Instead of values for `title` and `body`, the resolver for `addPlaylist` will now receive an object containing values for `input`. The following code reflects this change:

code/6-databases/src/resolvers.2.ts

```
67 async function addPlaylist(
68   _: any,
69   { input }: any,
70   { knex }: Context,
71 ): Promise<any> {
72   if (!knex) throw new Error('Not connected to the database');
73
74   const { title, body } = input;
```

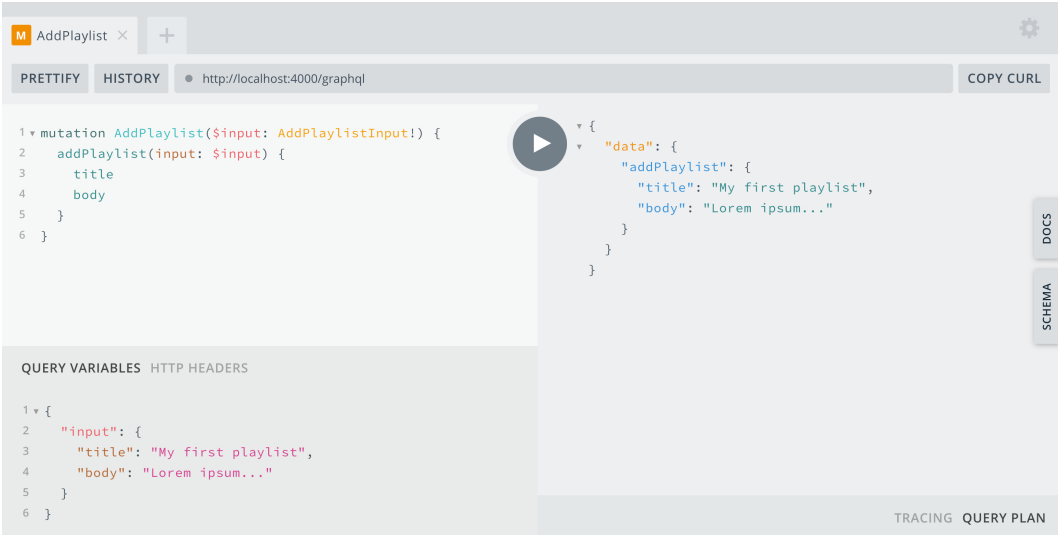
When sending a document with this mutation to the GraphQL server, you can now use the `AddPlaylistInput` type for the input:

```
mutation AddPlaylist($input: AddPlaylistInput) {
  addPlaylist(input: $input) {
    title
    body
  }
}
```

And use the following value for the `input` argument:

```
{
  "input": {
    "title": "My first playlist",
    "body": "Lorem ipsum..."
  }
}
```

This mutation requests the `title` and `body` fields as a response, as specified in the schema. However, looking at the schema, all fields from the `Playlist` type could be requested, including the tracks that are added to this playlist. As this playlist has just been created, there wouldn't be any tracks in it. Although this mutation is used to create a new instance of `Playlist`, it doesn't mean that all fields from that type are available as you can see from the response:



The GraphQL playground with a named mutation

Generally, this problem can be solved by creating a payload type, which is a regular GraphQL type used for the response of a mutation. The payload type only returns the fields that can be added (or edited) with this mutation, and is used instead of the actual type that is being mutated:

code/6-databases/src/schema.4.ts

```
34  type AddPlaylistPayload {
35    id: ID!
36    title: String!
37    body: String
38  }
39
40  type Mutation {
41    addPlaylist(input: AddPlaylistInput!): AddPlaylistPayload
42  }
```

The resolver for the `addPlaylist` mutation was already returning `null` for the field `artists`, but now your schema is also aware of the fields that are returned from this mutation when executed.

Handling Errors

You've briefly seen how GraphQL throws errors, such as when you're using wrong input fields, or when data is unavailable.

In another example, if you send a query to retrieve playlists, and a requested `id` doesn't exist, GraphQL will return an error. To explore GraphQL errors in more detail, let's create a new mutation to add a track to a playlist. This mutation would need both a track `id` and an `id` for the playlist that we want to add the track to:

code/6-databases/src/schema.5.ts

```
40  input SaveTrackInput {
41    playlistId: ID!
42    trackId: ID!
43  }
44
45  type SaveTrackPayload {
46    playlistId: ID!
47    playlistTitle: String!
48    trackId: ID!
49    trackTitle: String!
50  }
51
52  type Mutation {
53    addPlaylist(input: AddPlaylistInput!): AddPlaylistPayload
54    saveTrackToPlaylist(input: SaveTrackInput): SaveTrackPayload
55  }
```

The new `saveTrackToPlaylist` function takes both IDs and returns information about both the playlist and the track added. For this mutation to work, we need a new resolver that inserts IDs for the playlist and the track to the table `playlists_tracks`. This resolver also checks if the playlist and the track exist in the database, and if they don't, it throws an error:

code/6-databases/src/resolvers.5.ts

```
async function saveTrackToPlaylist(
  _: any,
  { input }: any,
  { knex }: Context,
): Promise<any> {
  if (!knex) throw new Error('Not connected to the database');

  const { playlistId, trackId } = input;

  const playlist = await knex('playlists').where('id', playlistId).sele\
ct();
  const track = await knex('tracks').where('id', trackId).select();

  if (!playlist.length) throw new Error('Playlist not found');
  if (!track.length) throw new Error('Track not found');

  await knex('playlists_tracks').insert({
    playlist_id: playlistId,
    track_id: trackId,
  });
```

This function will call the database three times. It executes two queries to get the results from the `tracks` and `playlists` tables, and a third query that inserts the information from the GraphQL input arguments into the database table `playlists_tracks`:

```
SELECT * FROM `playlists`;
SELECT * FROM `tracks`;
INSERT INTO `playlists_tracks` (`playlist_id`, `track_id`) VALUES (`pla\
ylist_id`, `track_id`);
```

Also, the function `saveTrackToPlaylist` must be added to the resolver-map to be able to add the tracks to a playlist using a mutation:

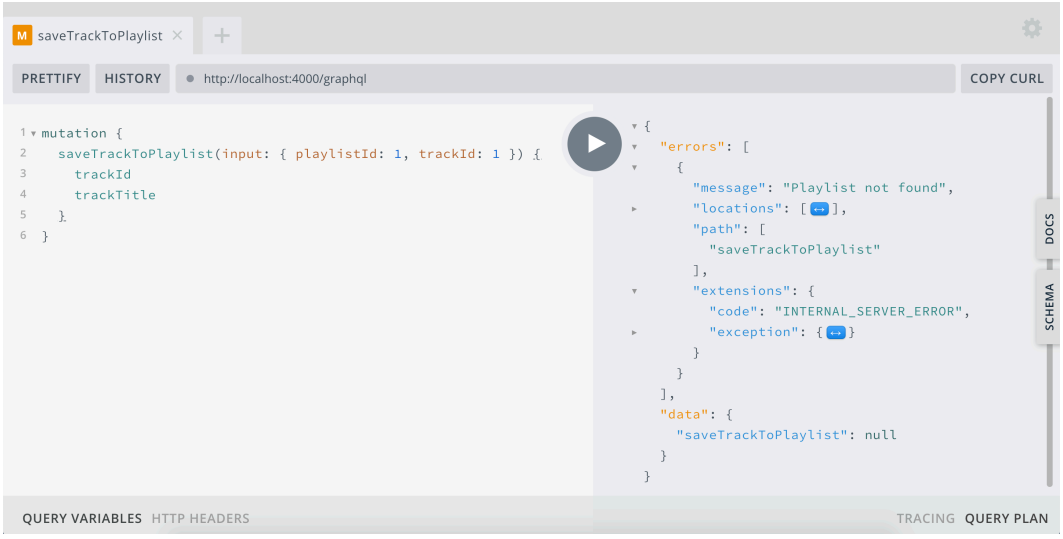
code/6-databases/src/resolvers.5.ts

```
124   Mutation: {
125     addPlaylist,
126     saveTrackToPlaylist,
127   },
```

When it throws an error, the resolver uses the `Error` object, and the errors that it returns are then added to the `error` object of the GraphQL response. If you haven't added any playlists yet, the following mutation won't resolve:

```
mutation {
  saveTrackToPlaylist(input: { playlistId: 1, trackId: 1 }) {
    id
    title
  }
}
```

In addition to a `data` object that always returns `null` for this mutation, GraphQL server's response will also include an `error` object of the following shape that you can see in this screenshot of the playground:



The GraphQL playground with an error response

The snippet above has been truncated but the actual response in the GraphQL playground is a huge chunk of JSON, which makes it hard to get the important information from that response.

While the fields `message`, `locations` and `path` help investigate the cause of an error, often you want to receive more information. Fortunately, the GraphQL specification allows you to add a field called `extensions` to the `error` object. Use this field to add any additional data that you want to share with the client application using the GraphQL server.

To extend the `error` object, you can either throw a custom `Error` object or use predefined error methods available in Apollo Server. Apollo Server provides several predefined errors, such as `AuthenticationError`, `ForbiddenError`, and `UserInputError`, as well as the general `ApolloError`. These errors make it easier for you to debug and read errors from the GraphQL server.

Let's update the resolver file and import `UserInputError` from `apollo-server-express`:

code/6-databases/src/resolvers.6.ts

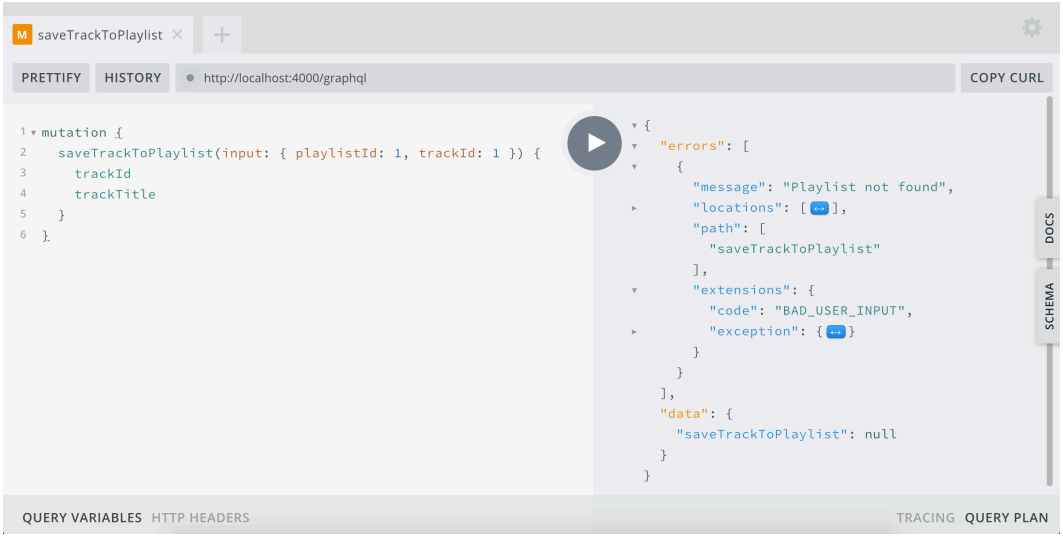
```
1  import { UserInputError } from 'apollo-server-express';
```

And use it to construct the error when a playlist or track is missing:

code/6-databases/src/resolvers.6.ts

```
92    if (!knex) throw new Error('Not connected to the database');
93
94    const { playlistId, trackId } = input;
95
96    const playlist = await knex('playlists').where('id', playlistId).sele\
97  ct();
98    const track = await knex('tracks').where('id', trackId).select();
99
100   if (!playlist.length) throw new UserInputError('Playlist not found');
101   if (!track.length) throw new UserInputError('Track not found');
```

If you look at the code above, not much has changed except for the shape of the `errors` object. If we send a document with this mutation to the GraphQL server and provide `id` for a non-existing playlist or track, this will again throw an error, but with a different `code` in the `extensions` field of the `errors` object:



The GraphQL playground with a customized error response

Before our latest change to the resolver, this error caused by wrong user input was defined vaguely as an `INTERNAL_SERVER_ERROR`. By returning `BAD_USER_INPUT` instead, you can use this code in your client application to display the error in the correct form to the user.

Handling errors this way works especially well when you're using Apollo to create your GraphQL server, but there are more declarative ways of returning errors to the client. One way of doing so is by adding errors as data. This approach has several advantages:

- you no longer have to return `null` for the payload of the mutation,
- errors become available in the GraphQL schema, and you can include them in the response.

To add errors to your data, you need to use the `Union` type (a.k.a. `Result`) in your GraphQL schema. The current payload type `SaveTrackPayload` becomes the success payload, and you should rename it accordingly. You will then combine it with new payload types when errors occur:

code/6-databases/src/schema.6.ts

```
type SaveTrackSuccess {
  playlistId: ID!
  playlistTitle: String!
  trackId: ID!
  trackTitle: String!
}

type SaveTrackPlaylistError {
  playlistId: ID!
  message: String!
}

type SaveTrackError {
  trackId: ID!
  message: String!
}

union SaveTrackPayload =
    SaveTrackSuccess
  | SaveTrackPlaylistError
  | SaveTrackError
```

The return payload for `saveTrackToPlaylist` remains the same, so you don't have to change anything to the `Mutation` type in the GraphQL scheme definition.

code/6-databases/src/schema.6.ts

```
type Mutation {
  addPlaylist(input: AddPlaylistInput!): AddPlaylistPayload
  saveTrackToPlaylist(input: SaveTrackInput): SaveTrackPayload
}
```

As the payload doesn't change, the `saveTrackToPlaylist` will still return data of the payload type `SaveTrackPayload` but the data can now take shape of `SaveTrackSuccess`, `SaveTrackPlaylistError`, or `SaveTrackError`. We also need to make several changes in the resolver because it currently throws an error if a playlist or track is not found:

code/6-databases/src/resolvers.7.ts

```
  if (!playlist.length)
    return {
      resolveType: 'SaveTrackPlaylistError',
      playlistId,
      message: 'Playlist not found',
    };

  if (!track.length)
    return {
      resolveType: 'SaveTrackError',
      trackId,
      message: 'Track not found',
    };
```

When a track is added to a playlist, the resolver returns the same information about the playlist and the track as before. However, when it is unable to find a playlist or track, the resolver now returns an object with the missing `id`, a message, and a `resolveType`. The latter is important because Apollo Server needs to know the shape of data returned by the resolver, and the `resolveType` field in the resolver-map helps choose a specific shape:

code/6-databases/src/resolvers.7.ts

```
  SaveTrackPayload: {
    __resolveType(obj: any) {
      if (obj.resolveType) return obj.resolveType;

      return 'SaveTrackSuccess';
    },
  },
```

The combined payload result for the `saveTrackToPlaylist` mutation gets an additional field called `resolveType`. The value for this field is passed by the resolver for `saveTrackToPlaylist`, or set to `SaveTrackSuccess` if it's not defined. You can now request error or success payload results when adding a track to a playlist. To do this, pass the following mutation in a document to the GraphQL server:

```
saveTrackToPlaylist(input: { playlistId: 1, trackId: 1 }) {
  ... on SaveTrackSuccess {
    playlistId
    playlistTitle
    trackId
    trackTitle
  }
  ... on SaveTrackPlaylistError {
    playlistId
    message
  }
  ... on SaveTrackError {
    trackId
    message
  }
}
```

This mutation uses fragments to request different response formats depending on which `resolveType` the resolver for `saveTrackToPlaylist` returns. Treating errors as data makes handling errors from a client application more sophisticated because you no longer need to extract the actual error from the `errors` object. This approach also lets you deal with errors in your schema, which means the data format is introspected and automatically documented by GraphQL as you can see in the playground.

The GraphQL playground with documented union response types

> You can add even more validations: for example, throw an error when a track has already been added to a playlist, or when a playlist with a given title already exists.

Summary

In this chapter, you have explored mutations, inserted data into a SQL-based database, and learned about error handling in GraphQL. You can apply the principles

covered in this chapter to other databases like MySQL, PostgreSQL, or even NoSQL databases. You may need to replace the Knex.js query builder with a different logic to retrieve and update data from a data source, but the approaches to querying and mutating data can be reused.

In the upcoming chapter, you will learn to deal with optimizations like batching and caching your data.

Caching and Batching

One thing about GraphQL servers that we haven't addressed so far in this book is performance optimization. When building any server (or application), you want the server to perform its tasks not only well but also fast.

When dealing with GraphQL servers, there are numerous optimizations you can implement, such as *caching* to return persistent data and *batching* to reduce the number of round trips your server makes to the database.

In this chapter, you'll learn more about the following optimizations: - Optimized queries - Caching - Batching - Cost computation

Optimized queries

When using GraphQL together with a data source, you want to limit the cost of queries in the documents that you send to your GraphQL server. In the previous chapters, you have written query logic but haven't implemented any optimizations to that logic.

One important optimization opportunity is to only retrieve from a data source that is defined in your query. By making use of field resolvers over query resolvers, you've already effectively optimized the query that retrieves tracks:

```
query {
  tracks {
    id
    title
    artist {
      firstName
    }
  }
}
```

Artist data is retrieved by the `getArtist()` resolver, and it's only called when artist details are defined in query fields. Whether or not the user is requesting the `title` of a track, the logic to get track information returns the entire instance of the track. GraphQL supplies the AST of the query as the fourth argument to every resolver, which you can use to get the fields that are requested.

So far we've covered three arguments of resolvers, leaving the fourth argument, the `info` object, out of scope. The `info` object defines the AST of the GraphQL server and provides additional information, such as all variables and operations.

The `info` object is of type `GraphQLResolveInfo` that you can import from the `graphql` package:

code/7-caching-batching/src/resolvers.1.ts

```
1 import { GraphQLResolveInfo } from 'graphql';
```

The structure of the `info` object is quite complex, and it will be easier to figure out if you log the `info` object for the `getTracks()` resolver:

code/7-caching-batching/src/resolvers.1.ts

```
async function getTracks(
  _: any,
  {},
  { knex }: Context,
  info: GraphQLResolveInfo,
): Promise<any> {
  if (!knex) throw new Error('Not connected to the database');

  console.log(info);

  return await knex('tracks').select();
}
```

In the code snippet above you can see that the type declaration for the `info` object is coming from the main GraphQL package and has the following structure:

- `fieldName` contains the name of a field that belongs to the current resolver.
- `fieldNodes` is an array containing extra information about the resolver's operation, including requested fields, arguments, and directives.
- `returnType` is a GraphQL type for this resolver.
- `parentType` is the GraphQL type for the parent object.
- `path` is the route to get fields for this resolver, based on the circularity of GraphQL.
- `schema` is the complete GraphQL schema for this resolver.
- `fragments` is a map of fragments used for this resolver's operation.
- `operation` is the previously mentioned AST for the used operation.
- `variableValues` is a map of variables that were provided for this resolver's operation.

These fields on the `info` object are related either to a specific field of the GraphQL operation or globally to the server. To test this, pass a document with the query above to the GraphQL server using the playground, and look at the terminal:

```
{
  fieldName: "tracks",
  fieldNodes: [
    // all the fields requested in this operation
  ],
  returnType: '[Track]',
  parentType: 'Query',
  path: { prev: undefined, key: 'tracks' },
}
```

Fields that are specific to this query and resolver include `fieldName` with the value of `tracks`. This entry corresponds to the top-level field of the query and has the `returnType` of `Track`. The `parentType` for `tracks` is the global operation type `Query`. The `path` is empty because there are no circular dependencies.

The other values - `schema`, `fragments`, `rootValue`, `operation`, and `variableValues` - are global and will be the same for every field in this resolver.

You can double-check this by also logging the result for the `info` object on the `getArtist()` resolver that retrieves artist details for a given track:

code/7-caching-batching/src/resolvers.1.ts

```
24 async function getArtist(
25   { artist_id: artistId }: any,
26   {},
27   { knex }: Context,
28   info: GraphQLResolveInfo,
29 ): Promise<any> {
30   if (!knex) throw new Error('Not connected to the database');
31
32   console.log(info);
33
34   return await knex('artists')
35   .where('id', artistId)
36   .select()
37   .first()
38 }
```

The contents of this object will be different from its parent resolver, `getTracks()`:

```
{
  fieldName: "artist",
  fieldNodes: [
    // all the fields requested in this operation
  ],
  returnType: 'Artist',
  parentType: 'Track',
  path: { prev: { prev: [Object], key: 0 }, key: 'artist' },
}
```

One of the reasons why developers choose to use GraphQL is they only want to receive the data they ask for. However, if you're querying all the fields in your table, this improvement will only make a difference for the client sending requests to your server. To take advantage of this on the server, resolvers can use the `fieldNodes` property to get information about which fields were requested.

Looking at the value of `fieldNodes`, we can see that its contents are deeply nested. To get all the requested fields from the `info` object at once, let's use the `graphql-list-fields` package. You can install `graphql-list-fields` and its type declarations with npm or Yarn; just make sure to include the `lodash` package that you'll also need:

```
npm i graphql-list-fields @types/graphql-list-fields lodash/snakeCase
```

Resolvers can use `graphql-list-fields` to minimize database queries by limiting them to the fields that were requested in a given operation.

code/7-caching-batching/src/resolvers.2.ts

```
2 import getFieldNames from 'graphql-list-fields';
```

After importing these packages, let's see what this would mean for the `getTracks()` resolver:

code/7-caching-batching/src/resolvers.2.ts

```
): Promise<any> {
  if (!knex) throw new Error('Not connected to the database');

  let fields = getFieldNames(info);
  fields = fields.map((fieldName) =>
    fieldName.startsWith('artist') ? 'artist_id' : fieldName,
  );

  return await knex('tracks').select('id', ...fields);
}

async function getArtist(
  { artist_id: artistId }: any,
  {},
  { knex }: Context,
```

The `getFieldNames()` method destructures all requested fields from the `info` object, including relational fields such as `firstName` of an artist. The result has the shape of `['id', 'title', 'artist.firstName']`. Each relational field (in this case `artist.firstName`) is replaced with a foreign key for the corresponding relation's table in the database.

The foreign key is needed for the `getArtist()` resolver that is called lazily when specified in a query to get tracks. Because field names in our schema are camel-cased but our database fields are snake-cased, we need to transform the field names provided by `info`. For this, we can use the `snakeCase` module from `lodash`. When we put everything together, our `getArtist()` resolver starts to look like this:

code/7-caching-batching/src/resolvers.2.ts

```
async function getArtist(
  { artist_id: artistId }: any,
  {},
  { knex }: Context,
  info: GraphQLResolveInfo,
): Promise<any> {
  if (!knex) throw new Error('Not connected to the database');

  let fields = getFieldNames(info);
  fields = fields.map((fieldName) => toSnakeCase(fieldName));

  return await knex('artists')
    .where('id', artistId)
    .select('id', ...fields)
    .first();
}
```

> The `graphql-list-fields` package also supports inline fragments, meaning your resolver code won't break if users use a different notation for their operations.

Using the `info` object helps you get insight into the execution of any operation and take advantage of the GraphQL server's AST, which provides great opportunities to optimize further. In the next section, you'll explore other caching and batching optimizations with DataLoader.

Batching

The optimizations described above are only the first step in getting the most out of GraphQL. In addition to only delivering the data that you requested, GraphQL can also solve the N+1 problem that was covered in a prior chapter and is typical of REST APIs.

To limit the number of round trips that an API makes to the database, one approach is to prevent making two identical database calls within a single request. Let's take a look at the query from the previous section:

```
query {
  tracks {
    id
    title
    artist {
      id
      firstName
    }
  }
}
```

What happens if there are multiple tracks by the same artist? Currently, the artist's information would be queried from the database on every track information request. To see for yourself, add the following `console.log()` statement to `src/resolvers.ts`:

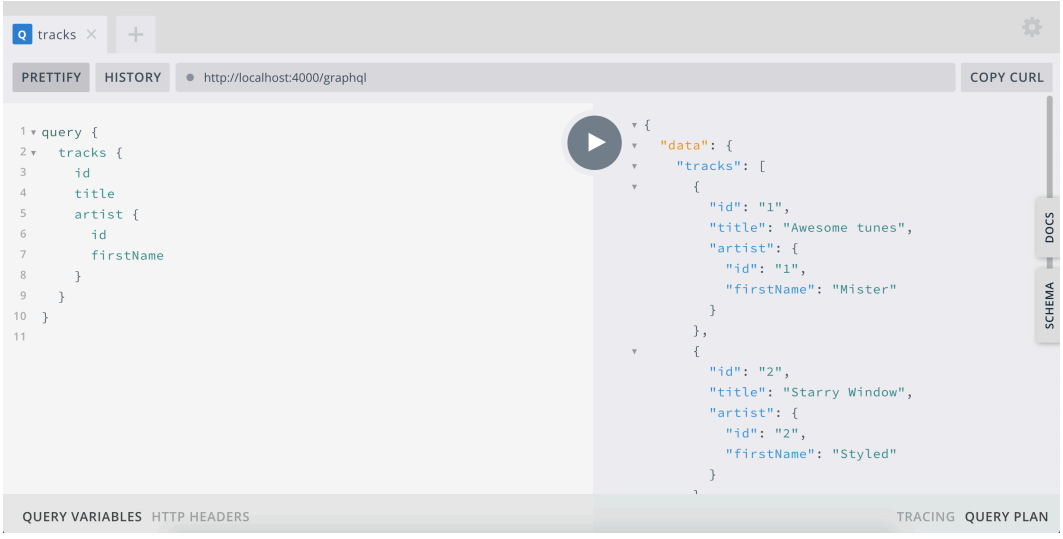
code/7-caching-batching/src/resolvers.3.ts

```
29 async function getArtist(
30   { artist_id: artistId }: any,
31   {},
32   { knex }: Context,
33   info: GraphQLResolveInfo,
34 ): Promise<any> {
35   if (!knex) throw new Error('Not connected to the database');
36
37   let fields = getFieldNames(info);
38   fields = fields.map((fieldName) => toSnakeCase(fieldName));
39
40   console.log('database called', artistId);
```

If you send a document with the query above and look at the output in the terminal, you'll see that the artist with the `id` of 2 is requested from the database twice:

```
database called 1
database called 2
database called 2
database called 3
```

The response of the GraphQL server to this query will be the same, no matter how many database calls you make.



The response as shown in the GraphQL playground

To introduce batching and prevent unnecessary database requests, we can use a utility called DataLoader. The `dataloader` package is available on npm and can be installed using npm or Yarn:

```
npm i dataloader
```

After installing DataLoader, you need to create actual dataloaders. These dataloaders will collect all IDs (or keys) of the information you need for an operation, and use this array of IDs to perform all database actions at once. Duplicate keys will be deleted, so the number of requests to your database will be kept at a minimum.

To implement the dataloader pattern, you first need to create a loader-map, which is an object that contains all dataloaders for your server:

code/7-caching-batching/src/index.1.ts

```
3  import DataLoader from 'dataloader';
4
5  import typeDefs from './schema';
6  import resolvers from './resolvers';
7  import startDatabase from './database';
8
9  const dataLoaders = async () => {
10   const db = await startDatabase();
11
12   return {
13     artist: new DataLoader((ids) => {
14       console.log('database called', ids);
15
16       return db('artists').whereIn('id', ids).select();
17     }),
18   };
19 };
```

Once this is done, add your dataloaders to the `context` object:

code/7-caching-batching/src/index.1.ts

```
21 const server = new ApolloServer({
22   typeDefs,
23   resolvers,
24   context: async () => {
25     const knex = await startDatabase();
26     const loaders = await dataLoaders();
27
28     return { knex, loaders };
29   },
30 });
```

Our loader-map contains a dataloader for `artist`. It creates a new `DataLoader` instance and queries the database for all provided IDs. The loader-map is created

asynchronously, as the database needs to be initialized as well. Note that database calls are logged.

You'll also need to make a few changes in the resolver to support the `artist` dataloader instead of working directly with the database object:

code/7-caching-batching/src/resolvers.4.ts

```
async function getArtist(
  { artist_id: artistId }: any,
  {},
  { loaders }: Context,
  info: GraphQLResolveInfo,
): Promise<any> {
  let fields = getFieldNames(info);
  fields = fields.map((fieldName) => toSnakeCase(fieldName));

  // go from snakecase to camelcase
  return await loaders.artist.load(artistId);
}
```

If you now try to send the same document as shown above to get tracks along with artist information, you'll see the following logged in your terminal:

```
database called [1, 2, 3]
```

From this response, you can see that not only requests are deduplicated, but requests to get different artists are in fact batched into one. Instead of executing four queries on your database, you're now querying just once and only with distinct IDs.

However, this response is not yet ready to be used by GraphQL, as data from the database needs to be transformed from snake case to GraphQL's native camel case. Let's transform the fields using the `camelCase` module from `lodash` that can be used in the same way as the `snakeCase` module shown in the previous section:

code/7-caching-batching/src/resolvers.5.ts

```
import { GraphQLResolveInfo } from 'graphql';
import getFieldNames from 'graphql-list-fields';
import toCamelCase from 'lodash/camelCase';
```

Using the `reduce()` method, we transform results from the dataloader to have camel-cased keys for fields.

You can perform similar optimizations with other resolvers and create more dataloaders. For example, in the next section about caching we'll optimize the `getTracks()` resolver.

code/7-caching-batching/src/resolvers.5.ts

```
async function getArtist(
  { artist_id: artistId }: any,
  {},
  { loaders }: Context,
): Promise<any> {
  const artists = await loaders.artist.load(artistId);

  return Object.keys(artists).reduce(
    (result, key) => ({
      ...result,
      [toCamelCase(key)]: artists[key],
    }),
    {},
  );
}
```

> Note that the `info` object is no longer used to optimize which fields are being queried, as this would require you to create separate dataloaders for every combination of fields that could be requested.

Caching

DataLoader is more than a great solution to handle batching: it can also solve situations where you need to query data from the database two or more times. With DataLoader, you can enable per-request caching that prevents overfetching data when you have a circular relation in your GraphQL operation.

One example of a circular relation is when you query for tracks in a playlist and their artists, but you also want to query all tracks:

```
query {
  tracks {
    title
    artist {
      id
      firstName
    }
  }
  playlist(id: 1) {
    tracks {
      id
      title
      artist {
        firstName
      }
    }
  }
}
```

Without using a dataloader for the `getArtist()` resolver, the tracks that are already returned for the top-level field `artists` will be fetched again when an ID occurs multiple times as a relation to `artists`. By utilizing a per-request caching solution like DataLoader, you can make sure that information about those artists is taken from a cache.

To try this yourself, add the query above to a document, send it to the GraphQL

server, and have a look at the response in the terminal. The dataloader for `artists` will be called:

```
database called [ 1, 2, 3 ]
```

> DataLoader uses a simple `Map` object for caching, and inserting very large responses into this object will affect the performance of your application. Therefore, this solution is not an alternative to other caching utilities like Redis or Memcached, but merely a way to optimize your resolvers on a lower level.

You don't have to use the DataLoader cache every time. For example, when you add (or delete) a track from a playlist, you want the playlist information to come directly from the database instead of DataLoader's per-request cache. Instead of adding this logic to the GraphQL server, you can add the following line of code that will delete IDs from the cache for `artists`:

code/7-caching-batching/src/resolvers.6.ts

```
30 async function getArtist(
31   { artist_id: artistId }: any,
32   {},
33   { loaders }: Context,
34 ): Promise<any> {
35   const artists = await loaders.artist.load(artistId);
```

To cache results longer than the per-request DataLoader cache does, you can apply other caching methods to GraphQL, such as HTTP caching or application-level caching. However, applying these methods to GraphQL, especially on the server side, could introduce certain constraints.

HTTP caching can be difficult for GraphQL servers, as not all requests will be returning the same content. HTTP caching works well when you need to cache GET requests to a uniquely identifiable endpoint. As you know, GraphQL only uses POST requests where the operation and the fields that you're passing in your document are

specified in the request body. HTTP caching is known to be inefficient when there are many large requests and is hard to do without global unique identifiers.

Application-level caching in GraphQL servers is usually implemented in either resolvers or the context, with the latter being the generally accepted location. When caching results in resolvers, you can use any caching method as the resolvers only contain logic to get data from a data source. Caching solutions that you can apply to the context or resolvers include popular tools like Redis or Memcached.

Cost computation

Fetching and computing data is expensive, and with GraphQL the "cost" of a query could, in theory, be infinite. Caching and batching provide just one solution to this problem, but another option could be to compute the cost of operations and limit it if it exceeds a certain threshold. Whether you want to implement a form or rate-limiting, protect the server from malicious Denial-of-Service attacks, or monetize your API, you can leverage the `info` object.

Consider the following query that our GraphQL schema could allow if you make small changes to it:

```
query {
  playlist(id: 1) {
    tracks {
      title
      artist {
        firstName
        tracks {
          artist {
            firstName
            tracks {
              artist {
                firstName
              }
            }
          }
```

```
          }
        }
      }
    }
  }
```

This query is heavily nested and has a lot of circular dependencies. When using DataLoader, this query probably wouldn't be a problem, but it certainly would in GraphQL servers that don't use batching and/or caching methods, or when users are requesting fields that cannot be taken from any cache.

One way to prevent this is by restricting GraphQL queries in the schema - specifically, by disallowing a nested circular dependency between `tracks` and `artists`. This is how the schema for our GraphQL server is designed right now, but if we add a reference to the type of `tracks` on the `Artist` type, we enable queries like the one shown above.

code/7-caching-batching/src/schema.ts

```
10  type Artist {
11    id: ID!
12    firstName: String!
13    lastName: String!
14    name: String!
15    tracks: [Track]
16  }
```

> The resolver-map for the code in this chapter has already been extended with a resolver to get the `tracks` field for the `Artist` type.

You're now able to query an infinite amount of circular references between tracks and artists. This query is going to be expensive for your database as the resolver that gets tracks for an artist has no caching or batching methods applied to it. Nesting these fields indefinitely, forcing the server to make a very expensive computation, can occur by accident or on purpose.

Using the `validationRules` field in Apollo Server options lets you inspect the query above. This is done by checking `ValidationContext` that contains information about operations and the schema, including the AST:

code/7-caching-batching/src/index.3.ts

```
21 const server = new ApolloServer({
22   typeDefs,
23   resolvers,
24   context: async () => {
25     const knex = await startDatabase();
26     const loaders = await dataLoaders();
27
28     return { knex, loaders };
29   },
30   validationRules: [
31     (val) => {
32       console.log(val);
33
34       return val;
35     },
36   ],
37 });
```

Using the validation context (`val`), you can calculate the cost of the AST for an operation sent to the GraphQL server and throw an error if the cost exceeds a certain threshold. Writing cost computation logic yourself can be non-trivial, as there are multiple ways to calculate the cost. Examples are statically checking the length (or depth) of an operation, or using factors to calculate operation complexity.

Instead, you can calculate the cost using the `graphql-query-complexity` library that is available on npm. Import this package in the file that contains the definition of your GraphQL server:

code/7-caching-batching/src/index.4.ts

```
import express from 'express';
import { ApolloServer } from 'apollo-server-express';
import DataLoader from 'dataloader';
import queryComplexity, { simpleEstimator } from 'graphql-query-complex\
ity';
```

By introspecting the AST and linking a factor cost to an operation using the `simpleEstimator()` method, the library calculates the cost of the operation based on the validation rules that you applied to your server configuration earlier:

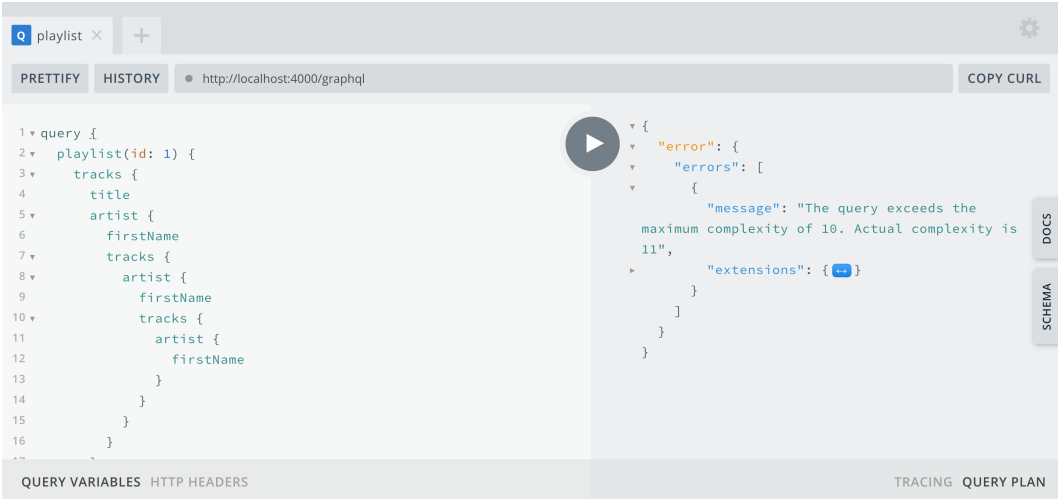
code/7-caching-batching/src/index.4.ts

```
  validationRules: [
    queryComplexity({
      estimators: [
        simpleEstimator({ defaultComplexity: 1 }),
      ],
      maximumComplexity: 10,
      onComplete: (complexity: number) => {
        console.log('Query Complexity:', complexity);
      },
    }),
  ],
```

If you run the complex query shown above in the GraphQL playground, the cost of this query would be `11`, which corresponds to the number of requested fields. This value is logged in the console as follows:

```
Query Complexity: 11
```

You can also use the `onComplete` callback to log this value somewhere else, store it in a database, or have a look at the error that the playground returns:



GraphQL query complexity output

`graphql-query-complexity` lets you create even more advanced cost computations: for example, you can use directives in the schema to set limits per type of operation.

Although this is a very simple example, it gives you a proper overview of possible scenarios where cost computation can improve the security of your GraphQL server.

Summary

In this chapter, you have created a scalable GraphQL server that used batching and per-request caching to improve performance. We have also covered the basics of cost computation for GraphQL operations. With these optimizations, you can seriously improve the performance of your GraphQL server and prevent users of your API from querying overcomplex queries.

The next chapter will show how further to improve your GraphQL server by adding authentication and authorization.

Authentication and Authorization in GraphQL

This chapter focuses on topics that often get neglected in GraphQL: authentication and authorization. GraphQL is all about getting data to the user, exactly as they have requested it. But what if the data (or parts of it) is not available to the requesting user? Let's see how to solve this problem by looking at the various ways to implement the authorization layer in GraphQL:

- Using resolvers
- Using the context
- With schema authentication using directives

JWT

Remember the GraphQL server with information about users and posts from chapter 4? You're going to build upon this server and add new (private) information to it.

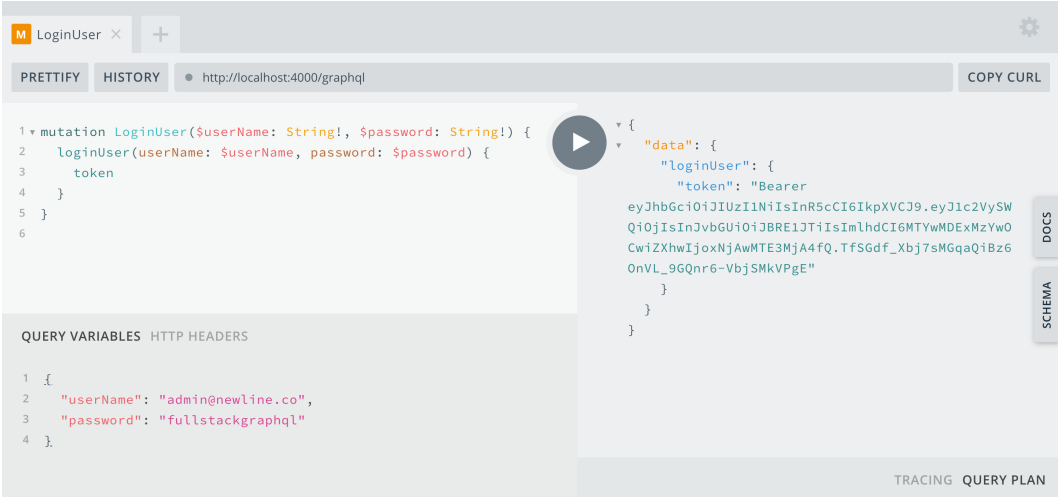
In this chapter, you'll use so-called JWTs (JSON Web Token), an open and industry-standard solution to authorization and information exchange, for authentication and authorization with GrahQL. You can request such a token by sending a document containing a mutation and the basic authentication (username and password) information for a user:

```
mutation LoginUser($userName: String!, $password: String!) {
  loginUser(userName: $userName, password: $password) {
    token
  }
}
```

With the following arguments:

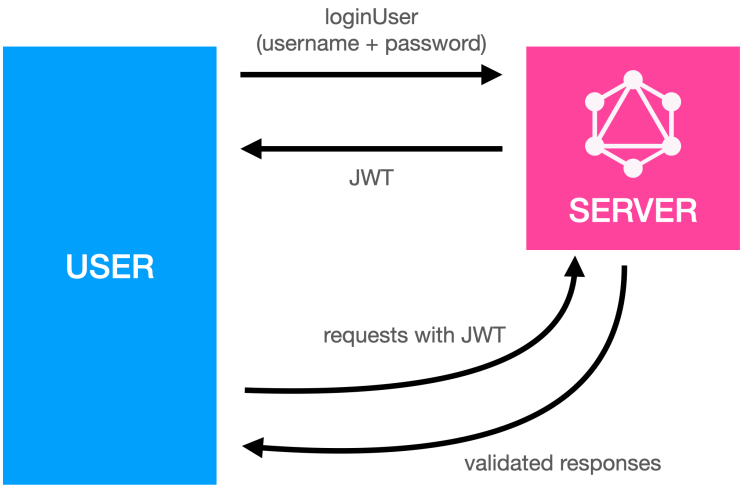
```
{
  "userName": "editor@newline.co",
  "password": "fullstackgraphql"
}
```

If you'd pass this document to the GraphQL server using the playground, the server will respond with the JWT if both the username and password are valid. This token will have the same format as you can see here:



Authentication with GraphQL

The authentication flow of JWT follows that of OAuth 2.0, which is currently one of the most widely used authentication standards for web applications. In short, OAuth 2.0 requires users to log in with their credentials from the client, and when they do, they receive a token in return. With this token, the client can request information from the server, and the server responds based on the information in that token.



OAuth 2.0 authentication flow

A JWT consists of three parts: header, payload, and signature. The header and the signature are used to securely encode token information for authorization, while the payload is used for information exchange. Whenever you pass a token to this GraphQL server, it will decode the JWT to get `userId` from the payload. A decoded JWT for the editor will hold the following payload:

```
{
  "userId": 1,
  "role": "EDITOR",
  "iat": 1587052134,
  "exp": 1587055734
}
```

As you can see, the payload of this JWT contains `userId` and `role` of the user along with data about the initialization and expiration of the token. JWTs primarily serve to authorize transactions between servers and/or clients and must never be used to encode private information.

Note: For more information on JWT, see https://jwt.io.

In this chapter, you'll create a server-side implementation of the 0Auth 2.0 flow, which you can then use to communicate with any web application that implements this protocol.

Resolver Authentication

The schema for our server defines fields for the type `Post`. Right now, if you send a document with a query for posts to the server, all these fields can be returned. Let's assume that some fields for this type should only be visible to authenticated users – that is, users who pass a user token along with their query:

code/8-authorization/src/schema.1.ts

```
  type Post {
    id: ID!
    title: String!
    body: String!
    author: User
    views: Int # this field should only be visible to admins
    published: Boolean! # unpublished posts are only visible to editors\
 and admins
  }
```

We have extended the `Post` type with two new fields: one defines if a post is published, and the other shows how often a post has been viewed.

To let your GraphQL server know that a user is authenticated, you can pass a token in HTTP headers along with your document containing the query. When you add this token to the context, you enable any resolver to validate if the user can retrieve certain information.

Apollo Server lets you get information from a request and add it to the context directly. To do this, add the context field to the options object when you create your GraphQL server:

code/8-authorization/src/index.1.ts

```
const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    const token = req.headers.authorization;

    console.log('token', token);

    return {
      token,
    };
  },
});
```

Any `authorization` HTTP header passed along with a request will now be added to the context and become available to your resolvers.

Your resolvers can now check if the token that the user has passed with a document is valid. Before you can read this value from the context, you need to define a type for the context so TypeScript doesn't throw any errors. You also need to import the function `isTokenValid` to validate the token:

code/8-authorization/src/resolvers.1.ts

```
import db, { Post, User } from './database';
import { isTokenValid, createUserToken } from './authentication';

type Context = {
  token: string;
};
```

To get a value for `token` from the context, you need to get the arguments from the `getPosts` resolver. In addition to context values, this resolver receives the parent object (defined as _) and GraphQL arguments:

code/8-authorization/src/resolvers.1.ts

```
14 function getPosts(_: Object, {}, { token }: Context): Array<Post> {
15   const posts = Array.from(db.posts.values());
16
17   if (!isTokenValid(token)) {
18     return posts.filter(({ published }) => published === true);
19   }
20
21   return posts;
22 }
```

The function `isTokenValid` in this example is validating the provided token. If the token is valid, it returns all posts from the database; otherwise, it only returns published posts.

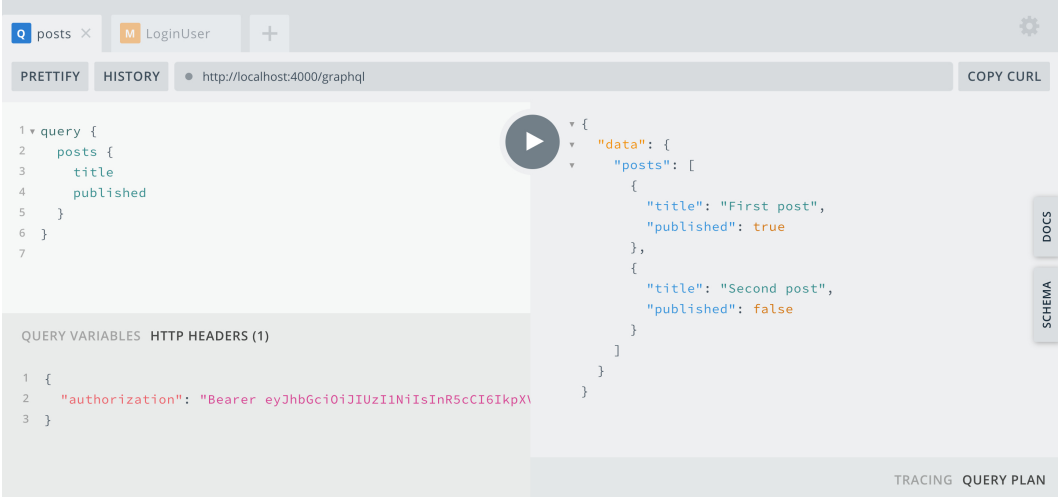
Go to the GraphQL Playground and try sending documents with and without the editor JWT that was created when you first started the GraphQL server:

```
query {
  posts {
    title
    published
  }
}
```

You can try sending a token with your document using the GraphQL Playground at http://localhost:4000/graphql. This token is the one that you created with the `loginUser` mutation in the first section of this chapter. In the left bottom side of the screen, you can pass query variables to use in a document, as well as HTTP variables in the following format:

```
{
  "authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2V\
ySWQiOjEsInJvbGUiOiJFRElUT1IiLCJpYXQiOjE1ODgzNDU5MDgsImV4cCI6MTU4ODM0OT\
UwOH0.lkUyyFjp6eyhPIVqwp08XU0fC5fd0CBJvBmxu8bbe4s"
}
```

When you don't pass the token, the response will only contain posts with `published` set to `true`. When you do pass the token, you'll see posts with `published` set to either `true` or `false` as you can see in the example below.



Passing authorization headers with GraphQL

When using resolvers for authentication, you may find yourself rewriting the same token validation logic over and over again. To address this problem, you can:

- use the context for authentication,
- add authentication validation to your schema, or
- write custom middleware to handle authentication.

  Note: If you're interested in using middleware to validate your GraphQL resolvers, the best approach would be to use a package like `graphql-shield`[29]. This package offers a set of higher-order functions and a middleware generator to pass authorization logic into your resolvers.

[29] https://github.com/maticzav/graphql-shield

Context Authentication

Instead of passing a token to your context on every request, you can choose to check if the token is valid and pass the authentication state of the user:

code/8-authorization/src/index.2.ts

```
import { isTokenValid } from './authentication';

const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    const token = req.headers.authorization;

    return {
      token,
      isAuthenticated: !!isTokenValid(token || ''),
    };
  },
});
```

This will enable your resolvers to read the value of `isAuthenticated` from the context. But first, you need to extend the context type to prevent TypeScript compiler errors.

code/8-authorization/src/resolvers.2.ts

```
type Context = {
  token: string;
  isAuthenticated: boolean;
};
```

After doing so you can modify the `getPosts` resolver to decide which posts should be returned based on the value of `isAuthenticated` from the context:

code/8-authorization/src/resolvers.2.ts

```
function getPosts(_: Object, {}, { isAuthenticated }: Context): Array<P\
ost> {
  const posts = Array.from(db.posts.values());

  if (!isAuthenticated) {
    return posts.filter(({ published }) => published === true);
  }

  return posts;
}
```

If you now send a document with the same query as in the previous section, you'll get the same results:

- if you pass the editor JWT, you'll receive all posts;
- without a token, you'll only receive published posts.

The validation logic doesn't need to be duplicated anymore, which means a lot less overhead when adding authentication rules to your GraphQL server.

In addition to authentication, you may need to apply role-based authorization to your GraphQL server. If you look at the schema for this chapter, notice it has a new field, `views`, that should only be available to users with admin privileges:

code/8-authorization/src/schema.1.ts

```
  type Post {
    id: ID!
    title: String!
    body: String!
    author: User
    views: Int # this field should only be visible to admins
    published: Boolean! # unpublished posts are only visible to editors\
 and admins
  }
```

Some of the fields in this schema, including `views`, are nullable: if data is not available, they will return `null` instead of throwing an error.

To check if a user is an admin, you need to extend the schema to allow the `User` type to have different roles. To accomplish this you need to create an `enum` type in GraphQL, which is similar to an `enum` in TypeScript, and is used to create a pre-defined set of allowed values and type of `Role`:

code/8-authorization/src/schema.2.ts

```
4  enum Role {
5    ADMIN
6    EDITOR
7  }
```

The enum type `Role` takes care of typing user roles, and must be used as the type for the field `role` on the `User` type in your schema:

code/8-authorization/src/schema.2.ts

```
18 type User {
19   id: ID!
20   firstName: String!
21   lastName: String!
22   name: String!
23   age: Float
24   email: String
25   posts: [Post]
26   role: Role! # users are either admins or editors
27 }
```

By changing the context for your GraphQL server, you can now get a user and their role based on the token that comes with a request.

The function `isTokenValid` decodes the token supplied with a request and returns JWT payload that contains `userId` of the requesting user. Using this ID, you can retrieve the user's record from the database and add user details to the context:

code/8-authorization/src/index.3.ts

```
import db from './database';
import { isTokenValid } from './authentication';

const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    const token = req.headers.authorization;
    const { userId } = isTokenValid(token || '') || {};
    const user = db.users.get(userId);

    return {
      token,
      isAuthenticated: !!user,
      user,
    };
  },
});
```

As soon as the token is validated, the user is added to the context and should be added to the `Context` type for the resolvers.

code/8-authorization/src/resolvers.3.ts

```
type Context = {
  token: string;
  isAuthenticated: boolean;
  user: User;
};
```

You've already seen how that value can be retrieved from the context in resolvers. To make the field `views` for `Post` only visible to admin users, you need to filter results from the database to return `null` when the user is not an admin:

code/8-authorization/src/resolvers.3.ts

```
function getPosts(
  _: Object,
  {},
  { isAuthenticated, user }: Context,
): Array<Post> {
  const posts = Array.from(db.posts.values());

  if (!isAuthenticated) {
    return posts.filter(({ published }) => published === true);
  }

  return posts.map((post) => ({
    ...post,
    views: user.role === Role.Admin ? post.views : null,
  }));
}
```

You can now go to the GraphQL Playground and send a document with a query to retrieve posts, including the field `views`. If you pass the editor JWT or no token at all, the value for `views` will be `null`.

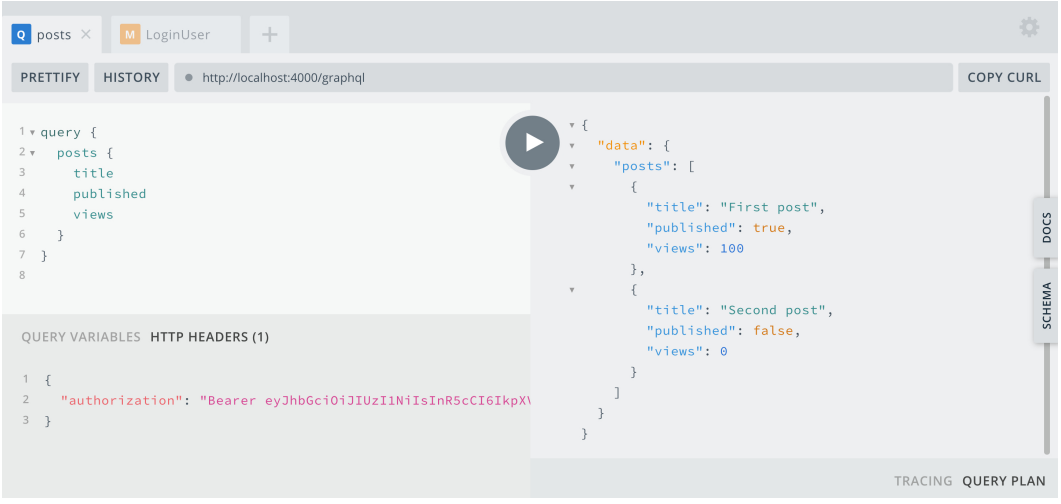
However, if you use a JWT for a user with admin rights, the GraphQL server will return a value for the field `views`. To create the JWT with the correct rights, you can use the following values when using the `loginUser` mutation:

```
{
  "userName": "admin@newline.co",
  "password": "fullstackgraphql"
}
```

After retrieving this token, you can pass it along with the query to retrieve the posts that you've used before including the field `views`:

```
query {
  posts {
    title
    views
  }
}
```

This will return the following output in the GraphQL playground, but only when a JWT for an admin user is set in the headers.



Passing along JWTs with different roles

The downside is that your users may not understand why the value for `views` is always `null` unless they are admin users. To solve this problem, you can return an error when the user requesting `views` is not an admin. Let's update the resolver `getPost` to throw an error with a message when the user doesn't have permissions to view the value for `views`:

code/8-authorization/src/resolvers.4.ts

```
27  return posts.map((post) => ({
28    ...post,
29    views:
30      user.role === Role.Admin
31        ? post.views
32        : new Error('Only admins can retrieve this field'),
33  }));
```

> Note: When using Apollo Server to create your GraphQL server, you can use the `AuthenticationError` class instead. This class provides a constructor that takes an error message and is specifically tailored to handle authentication errors.

When you now send a query to retrieve posts with the `views` field without passing an admin-level token, the response will contain an `error` object.

There's also a more sophisticated approach to let users know in advance that they're not allowed to query the field `views`. This approach to authentication involves the schema and directives.

Schema Authentication

So far, you have learned to handle authentication and authorization with resolvers and the context of your GraphQL server. Both approaches work fine, except that they neglect one of the core features of GraphQL – the schema.

Let's discard role-based authorization with resolvers and the context, and implement a new schema-based approach. Instead of resolvers, you'll need to add a custom directive to your schema.

Directives

Directives are an addition to a GraphQL schema that lets you define what should be returned by the server and what should not be returned. With directives, you can customize the response from the operation or the schema directly.

Before using directives in the schema, let's delete the logic that's validating the user token and the user role from the resolvers:

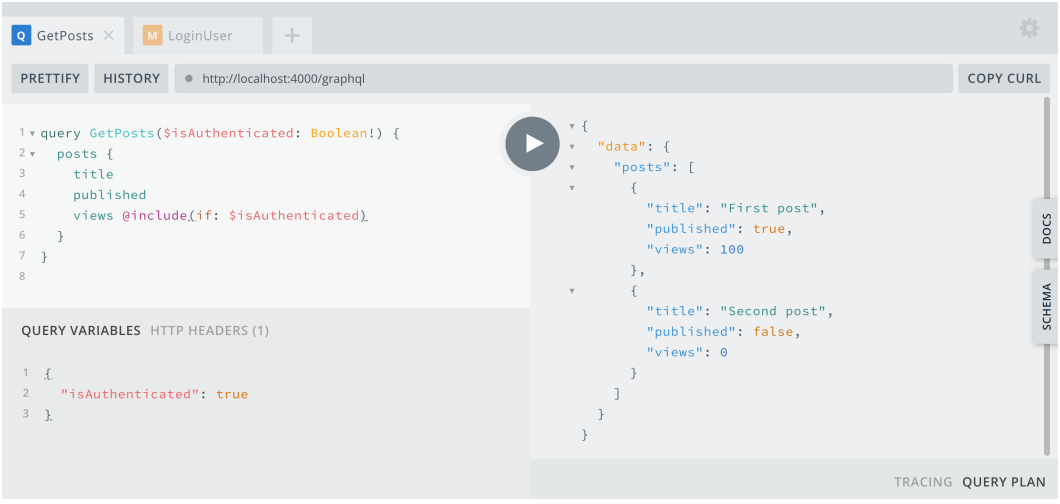
code/8-authorization/src/resolvers.5.ts

```
function getPosts(
  _: Object,
  {},
  { isAuthenticated }: Context,
): Array<Post> {
  const posts = Array.from(db.posts.values());

  if (!isAuthenticated) {
    return posts.filter(({ published }) => published === true);
  }

  return posts;
}
```

The GraphQL specification includes two possible directives, which are all supported by Apollo Server as well. According to this specification, all implementations should provide the `@skip` and `@include` directives, that you can use in any operation without having to explicitly define them in the schema.

For the `@include` directive this means that the field it's attached to will only be included when the `if` argument is true, as you can see in the screenshot below for the following query:

```
query GetPosts($isAuthenticated: Boolean!) {
  posts {
    title
    published
    views @include(if: $isAuthenticated)
  }
}
```



Using the @include directive in the playground

The `@skip` directive is doing the exact opposite, as you can see when using this query that skips the field `views` when the boolean argument `$isAuthenticated` is true:

```
query GetPosts($isAuthenticated: Boolean!) {
  posts {
    title
    published
    views @skip(if: $isAuthenticated)
  }
}
```

> Apollo Server also supports the optional directive `@deprecated` that can be used to send information about the field that will be deprecated to the users of your GraphQL API. The `@depricated` directive is added in the schema instead of an operation.

Custom directives

The default directives already let you define which fields should or shouldn't be returned, however, the argument that determines this can be easily modified by the

user as there is no server validation. Therefore you must create a custom directive that's able to validate the JWT and based on that token will return the information the user is allowed to receive.

Before adding a custom directive to your schema, think about the problem you're trying to solve. In our scenario, we add a directive to define the authentication state and user roles in the schema. This makes the schema aware of what information is public or private. Here's how you can define this in the schema:

code/8-authorization/src/schema.3.ts

```
4   enum Role {
5     ADMIN
6     EDITOR
7   }
8
9   directive @auth(role: Role) on FIELD_DEFINITION
10
11  type Post {
12    id: ID!
13    title: String!
14    body: String!
15    author: User
16    views: Int @auth(isAuthenticated: true, role: ADMIN) # this field s\
17 hould only be visible to admins
18    published: Boolean! # unpublished posts are only visible to editors\
19  and admins
20  }
```

A new custom directive in the schema defines that the field `views` requires a user to be authenticated and have the role "admin", based on the enum type `Role` that you've added before. We still need to come up with logic to make sure that non-admin users can't see values for `views`.

In a new file called `directives.ts` in the `src` directory, let's create a new class to define the custom directive by adding the following code:

code/8-authorization/src/directives.1.ts

```
import { defaultFieldResolver } from 'graphql';
import { SchemaDirectiveVisitor } from 'apollo-server-express';
import { User } from './database';

export class AuthDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field: any) {
    const { resolve = defaultFieldResolver } = field;
    const { isAuthenticated, role } = this.args; // The arguments for t\
he directive

    field.resolve = async (
      root: Object,
      args: Object,
      context: { user: User },
      info: Object,
    ) => {
      const fieldValue = await resolve(root, args, context, info);

      return fieldValue;
    };
  }
}
```

This, however, is just the boilerplate code to add a custom directive to your GraphQL server. The class `AuthDirective` defines a method called `visitFieldDefinition`. The method will be called for every `FieldDefinition`, collecting information about all fields in the schema that use the `@auth` directive. To handle the validation, this should be extended with these two statements:

code/8-authorization/src/directives.2.ts

```
17        const { user } = context;
18
19        // Throw an error when there is no authenticated user
20        if (isAuthenticated && !user) {
21          return new Error(`Only authenticated users can access ${field.n\
22  ame}`);
23        }
24
25        // Throw an error when the user requests a field with the wrong r\
26  ole
27        if (user.role !== role) {
28          return new Error(
29            `Only users with the role ${role} can access ${field.name}`,
30          );
31        }
32
33        return fieldValue;
```

Based on the role of the user added to the context, it will return a value for every such field. If the user is not authenticated or not authorized to view a certain field, it will throw an error.

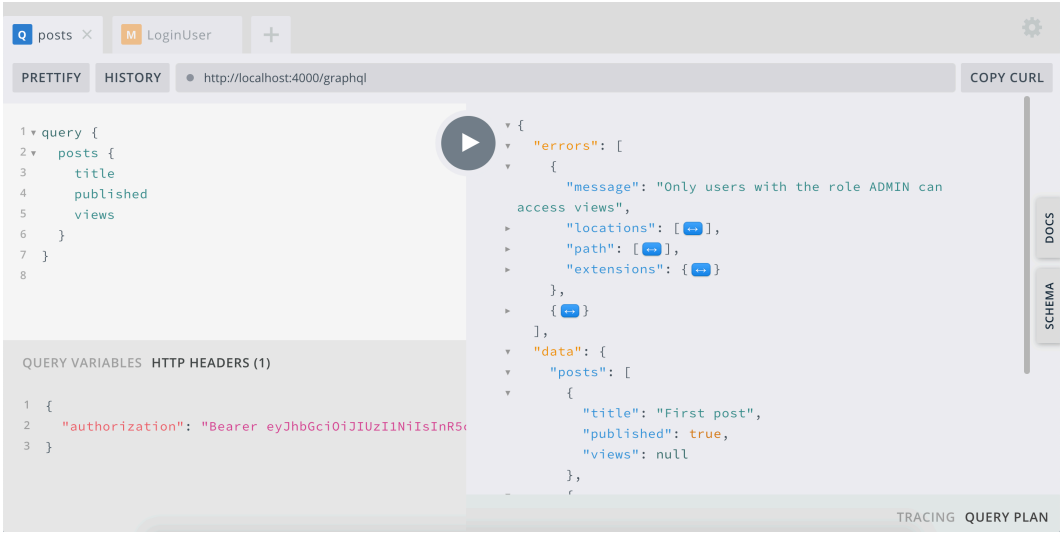
> Note: Only the `context` is strictly typed for the `resolve` function, as it is the only value you're using.

To add this directive to Apollo Server setup, import the directive into the file `src/index.ts`:

code/8-authorization/src/index.4.ts

```
8  import { AuthDirective } from './directives';
9
10 const server = new ApolloServer({
11   typeDefs,
12   resolvers,
13   context: ({ req }) => {
14     const token = req.headers.authorization;
15     const { userId } = isTokenValid(token || '') || {}
16     const user = db.users.get(userId);
17
18     return {
19       token,
20       isAuthenticated: !!user,
21       user,
22     };
23   },
24   schemaDirectives: {
25     auth: AuthDirective, // Logic for the directive @auth
26   },
27 });
```

When you use a query to get the posts in a document without a (valid) JWT for a user with admin rights, the GraphQL server will respond with both an error message and the partial result of the query if you request the field `views`.



Using the custom @auth directive in the playground

The same approach can be used to define what fields users can edit in a mutation.

Mutations and authorization

In addition to limiting the fields that are returned on queries, directives can be applied to mutations. The mutation `addPost` should only be available to users that are authenticated as either editors or admins. In your schema, add a new mutation along with input arguments:

code/8-authorization/src/schema.4.ts

```
35  input PostInput {
36    id: ID!
37    title: String!
38    body: String!
39  }
40
41  type Query {
42    users: [User]
43    posts: [Post]
44  }
```

```
45
46    type Mutation {
47      addPost(post: PostInput): Post @auth(isAuthenticated: true)
48      loginUser(userName: String!, password: String!): Credentials
49    }
```

This new mutation takes three arguments and is only available to authenticated users. User token validation still occurs both in the context that contains the user object and the class for the `@auth` directive. You should now add a resolver for the `addPost` mutation that receives post info and the context as arguments:

code/8-authorization/src/resolvers.6.ts

```
40  function addPost(
41    _: Object,
42    { post }: { post: PostInput },
43    { user }: Context,
44  ): Post | Error | undefined {
45    if (user) {
46      if (db.posts.get(post.id)) {
47        return new Error('A post with this id already exists');
48      }
49
50      const newPost = new Post(post.id, user.id, post.title, post.body, f\
51  alse, 0);
52
53      db.posts.set(post.id, newPost);
54
55      return newPost;
56    }
57  }
```

First, the resolver checks if a post with the provided `id` already exists, and if not, it adds the post to the database. The post that has been added to the database is returned to the user. To make this change effective, you'll also need to update the resolver-map with the resolver `addPost` for the mutation:

code/8-authorization/src/resolvers.6.ts

```
70   Mutation: {
71     addPost,
72     loginUser: createUserToken,
73   },
```

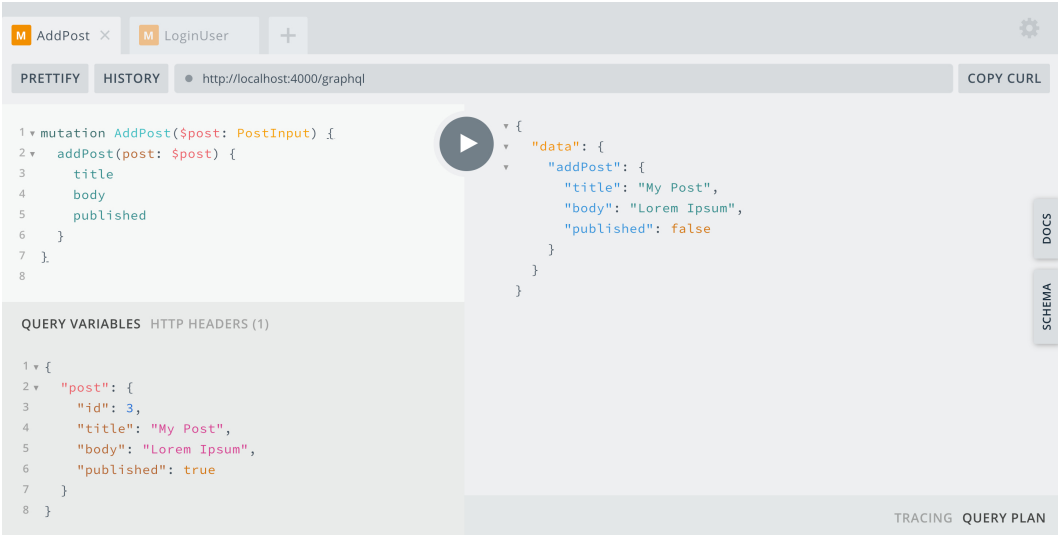
When you now add a new post using the `addPost` mutation, that has this format and returns the newly added post if succesfull:

```
mutation AddPost($post: PostInput) {
  addPost(post: $post) {
    title
    body
    published
  }
}
```

And these values that should be added to the database:

```
{
  "post": {
    "id": 3,
    "title": "My Post",
    "body": "Lorem Ipsum",
    "published": true
  }
}
```

If you look at the response you might notice that the post has the field `published` set to `false`, as you can see in this screenshot below:



Using the custom @auth directive in the playground

This is because you only want admins to be able to publish posts to the blog, something that should be clear from your schema. To make this possible, the directive `@auth` must accept `input` fields:

code/8-authorization/src/schema.5.ts

```
  directive @auth(
    isAuthenticated: Boolean!
    role: Role
  ) on FIELD_DEFINITION | INPUT_FIELD_DEFINITION
```

After this you can add the directive to `PostInput`:

code/8-authorization/src/schema.5.ts

```
  input PostInput {
    id: ID!
    title: String!
    body: String!
    published: Boolean! @auth(isAuthenticated: true, role: ADMIN) # thi\
s field should only be modifiable by admins
  }
```

In the updated schema above, the input type for posts gets a new field called `published`. Only admins should be able to modify this field.

In addition, the directive `@auth` now applies to input field definitions. This requires you add a new method, `visitInputFieldDefinition`, to the directive class `AuthDirective`. This method can contain the same code as the `visitFieldDefinition` method:

code/8-authorization/src/directives.3.ts

```
  visitInputFieldDefinition(field: any) {
    const { resolve = defaultFieldResolver } = field;
    const { isAuthenticated, role } = this.args; // The arguments for t\
he directive

    field.resolve = async (
      root: Object,
      args: Object,
      context: { user: User },
      info: Object,
    ) => {
      const fieldValue = await resolve(root, args, context, info);
      const { user } = context;

      // Throw an error when there is no authenticated user
      if (isAuthenticated && !user) {
        return new Error(`Only authenticated users can access ${field.n\
ame}`);
```

```
53        }
54
55        // Throw an error when the user requests a field with the wrong r\
56  ole
57        if (role && user.role !== role) {
58          return new Error(
59            `Only users with the role ${role} can access ${field.name}`,
60          );
61        }
62
63        return fieldValue;
64      };
65    }
```

With this last addition, only admin users can set the field `published` to `true`. All other users will get an error response notifying them that only users with the role "ADMIN" can access the field `published` when adding a post.

You have now implemented both authentication and role-based authorization for your GraphQL server.

Not too bad, right? Adding custom directives to your schema makes it more readable and improves the usability of your GraphQL server. Directives add a declarative layer for authentication and authorization and respect GraphQL standards.

Writing custom directives and logic for every field you want to exclude for certain users can be a daunting task. Fortunately, there are packages like `graphql-shield`[30] to help you out. These implementations can be used with any GraphQL server, and by applying the OAuth 2.0 standard you can add authorization to your client application.

Summary

From reading this chapter you've learned how to add authentication and authorization using JWTs to a GraphQL server. Multiple approaches are described and discussed, which are adding the logic to the resolvers, context, or even to the

[30] https://github.com/maticzav/graphql-shield

GraphQL schema with custom directives. By using JWTs you can make distinctions between user roles and permissions, to determine what information is available to the user.

The next chapter will focus on creating type-safe servers directly from type definitions using a code-first approach to creating GraphQL servers.

Code First GraphQL with TypeORM and TypeGraphQL

So far we've learned the prudent way of creating GraphQL servers by explicitly designing and writing a schema upfront.

However, this can be time-consuming and makes it complex to update your server when some of the data changes.

In this chapter we will explore a code-first approach where we infer the schema from our code and thus avoid potential out-of-sync issues between resolvers, schemas and typescript types.

There are a variety of ways to do this. Some of the popular packages for database-generated GraphQL servers are:

- Prisma[31]
- Hasura[32]
- Postgraphile[33]

All three of these platforms make opinionated choices about how your database should be structured, what your access patterns will be, how to handle authentication, and various other tradeoffs.

They're all good choices – if they match your use case and you're comfortable with their constraints.

You may, however desire for less magic than they offer and want to have the convenience of auto-generation - without taking the black-box approach of setting up another process in your infrastructure.

[31] https://www.prisma.io/
[32] https://hasura.io/
[33] https://www.graphile.org/postgraphile/

This chapter aims to teach you an approach where you're able to connect a Node.js ORM directly to your GraphQL server. This is a nice tradeoff between ease-of-use and low-level control.

After reading this chapter you will know how to:

- Create a code-first GraphQL server
- Use TypeORM to query databases with GraphqL
- Add authorization to a code first server

For this we will implement a simple blogging API to explore the TypeORM and TypeGraphQL packages.

TypeGraphQL

TypeGraphQL[34] is a library that automatically generates a GraphQL schema by introspecting your Typescript classes.

All you have to do to benefit from this wizardry is to sprinkle some TypeScript decorators.

This is how it looks:

code/9-typegraphql/simple/entities/Author.ts

```
17 @ObjectType()
18 export class Author {
19   @Field()
20   id!: number;
21
22   @Field()
23   name!: string;
24 }
```

[34] https://typegraphql.com/

This `@ObjectType`, `@Field` are decorators that annotate bake the necessary information into your class. This explicit API let's us have granular control over which fields become accessible in your GraphQL endpoint.

Next we need to write root Query resolver. In TypeGraphQL this takes form of a annotated class:

code/9-typegraphql/simple/entities/Author.ts

```
26 @Resolver(() => Author)
27 export class AuthorResolver {
28   @Query(() => [Author])
29   authors(
30     @Ctx() context: Context,
31     @Args() { offset, limit }: PaginationInput
32   ): Author[] {
33     return context
34       .authors
35       .slice(offset, offset + limit);
36   }
37
38   @FieldResolver()
39   posts(
40     @Ctx() context: Context,
41     @Root() author: Author,
42     @Args() {offset, limit}: PaginationInput
43   ): Post[] {
44     return context
45       .posts
46       .filter(post => post.authorId == author.id)
47       .slice(offset, offset+limit);
48   }
49
50   @Mutation(() => Author)
51   addAuthor(
52     @Ctx() context: Context,
53     @Arg("name") name: string
54   ): Author {
```

```
55      const author = {
56        id: context.authors.length,
57        name: name,
58      };
59
60      context.authors.push(author);
61
62      return author;
63    }
64  }
```

`@Resolver` specifies that this is a class for resolving authors and `Query` declares what type `authors` returns. For demonstration purposes, we just return a dummy user.

So far this defines the following schema:

```
type Author {
  id!: Int;
  name!: String;
}

type Query {
  authors: Author[]
}
```

Finally we hook it up with apollo-server to get our playground running:

code/9-typegraphql/simple/index.ts

```
import "reflect-metadata";
import { ApolloServer } from "apollo-server";
import { buildSchema } from "type-graphql";
import { AuthorResolver } from "./entities/Author";
import { PostResolver } from "./entities/Post";
import { createContext } from "./context";

async function main() {
  // build TypeGraphQL executable schema
  const schema = await buildSchema({
    resolvers: [AuthorResolver, PostResolver],
  });

  // create a context for graphql resolvers
  const context = createContext();

  // create Apollo Server
  const server = new ApolloServer({
    schema,
    context: () => context
  });

  // Start the server
  const { url } = await server.listen(4000);
  console.log(`Server is running, GraphQL Playground available at ${url\
}`);
}

main();
```

And here we go - our first `type-graphql` server is up and running!

Let's discuss the boilerplate:

`reflect-metadata` is a special package that required by `type-graphql` to enable runtime-type reflection.

Apart from that, we use the `type-graphql` `buildSchema` function to - as the name says - build our schema. It takes a list of resolvers and instantiates our resolver classes as well.

This is, of course, a not very exciting GraphQL server as it has no real functionality. So let's go back to our resolver and add a mutation to fill our in-memory database. To do that we need to extend our `AuthorResolver`:

code/9-typegraphql/simple/entities/Author.2.ts

```
34   addAuthor(
35     @Arg("name") name: string
36   ): Author {
37     const author = {
38       id: this._authors.length,
39       name: name
40     };
41
42     this._authors.push(author);
43     return author;
44   }
```

Similar drill. We declare `addAuthor` as a `@Mutation` that returns an `Author` and specify that we have a parameter `@Arg` called `name`.

And that is pretty much it, now we are able to fill the 'database' using a mutation:

```
mutation {
  addAuthor(name: "blub") {
    id
    name
  }
}
```

and then query our data via:

```
query { authors { id, name } }
```

As the list will quickly grow in size we'll need to implement some pagination logic.

Implementing Pagination

For that we add another entity called `PaginationInput`:

code/9-typegraphql/simple/entities/PaginationInput.ts

```
import { ArgsType, Field } from "type-graphql";

@ArgsType()
export class PaginationInput {
  @Field({ nullable: true, defaultValue: 0 })
  offset!: number;

  @Field({ nullable: true, defaultValue: 10 })
  limit!: number;
}
```

Our `PaginationInput` is declared as such via the `@ArgsType` decorator and consists of two fields `offset` and `limit`.

We then define `defaultValues` for each field and GraphQL will always guarantee that those values are not undefined.

We then add those to Our `AuthorResolver`:

code/9-typegraphql/simple/entities/Author.3.ts

```
  authors(
    @Ctx() context: Context,
    @Args() { offset, limit }: PaginationInput
  ): Author[] {
    return context.authors.slice(offset, offset + limit);
  }
```

The `@Args` decorator tells TypeGraphQL that we want to map the `PaginationInput` arguments to the first argument of our resolver.

Using Context in a Resolver

By now you should have noticed that the resolvers in this framework deviate from our the traditional function definitions of resolvers and you might wonder how to access context or other fields in the resolver.

The way this works is again via a decorator. If you need the context you simply add an argument `@Ctx()` decorator to your function. We can now move our `authors` to a context and decouple this from the resolver class.

code/9-typegraphql/simple/context.ts

```
import { Author } from "./entities/Author";
import { Post } from "./entities/Post";

export interface Context {
  authors: Author[];
  posts: Post[];
}

export const createContext = () => ({
  authors: [],
  posts: [],
});
```

This defines our `Context` type and a function that creates a context.

We now can use the context in our author resolver:

code/9-typegraphql/simple/entities/Author.ts

```
@Resolver(() => Author)
export class AuthorResolver {
  @Query(() => [Author])
  authors(
    @Ctx() context: Context,
    @Args() { offset, limit }: PaginationInput
  ): Author[] {
    return context
      .authors
```

This allows us to share our author database across resolvers.

We can appreciate this as soon as we have another entity with resolvers. This will allow us ultimately to join multiple datasets together. So let's add a `Post` entity:

code/9-typegraphql/simple/entities/Post.ts

```
@ObjectType()
export class Post {
  @Field()
  id!: number;

  @Field()
  title!: string;

  @Field()
  text!: string;

  authorId!: number;
}
```

Our `Post` type exposes `id`, `title` and `text` to GraphQL. However, `authorId` is a field to internally track to which author a post belongs:

code/9-typegraphql/simple/entities/Author.ts

```
50   @Mutation(() => Author)
51   addAuthor(
52     @Ctx() context: Context,
53     @Arg("name") name: string
54   ): Author {
55     const author = {
56       id: context.authors.length,
57       name: name,
58     };
59
60     context.authors.push(author);
61
62     return author;
63   }
```

Here we say that our field resolver `posts` requires context and the `Root` object of type `Author`. This is the first argument in traditional resolver functions.

Then in the function body we use `author.id` to find all posts with matching `authorId`.

This will ultimately enable us to query posts of each author:

```
query {
  authors {
    name
    posts {
      title
    }
  }
}
```

Then lastly we add a mutation `addPost` to complete our playground:

code/9-typegraphql/simple/entities/Post.ts

```
41   @Mutation(() => Post)
42   addPost(
43     @Ctx() context: Context,
44     @Arg("authorId") authorId: number,
45     @Arg("title") title: string,
46     @Arg("text") text: string
47   ): Post {
48     const post = {
49       id: context.posts.length,
50       authorId,
51       text,
52       title,
53     };
54
55     context.posts.push(post);
56
57     return post;
58   }
```

This is mostly identical to Authors resolver. Note that instead of using Args we can also just use `Arg` multiple times to extract each individual argument from GraphQL into our resolver - there is no particular benefit between the two approaches but I'd recommend that you should maintain consistency.

We could spent a lot of time perfecting this API but do a quick recap what we've seen so far:

TypeGraphQL forces us to write our TypeScript types and resolvers first and then uses our annotations to generate an appropriate GraphQL schema. T

his means those things are always in-sync and we have type-safety by design without the need of a GraphQL type generator.

Also we witnessed how magical decorators in TypeScript can be and TypeGraphQL is embracing them to the fullest. Next we'll take a look how to integrate it with a database via TypeORM package. This is where TypeGraphQL really shines.

So buckle up for the next section, more decorators are coming!

TypeORM

TypeORM is a Object Relation Mapper that allows you to create bindings between your objects and a database. It does so by leveraging type annotations to then generate the appropriate database queries.

As TypeORM uses the same mechanism as TypeGraphQL, it is a wonderful pairing and will ensure that not only your types, resolvers and schemas are always in-sync but also your database.

TypeORM + TypeGraphQL will ensure everything is described in one place!

We will only cover the minimal amount of TypeORM to highlight the hinted synergy but TypeORM has much more to offer and I strongly recommend a deep dive into it's documentation.

Tt not only has support for a plethora of databases but also supports versioning, migrations and the usage of multiple databases.

Without further redo, let's get our hands dirty. We'll start where we left of in the previous chapter but install `TypeORM` and `sqlite3` as our simple database.

Then we start annotating our types in a similar fashion as before:

code/9-typegraphql/with-typeorm/entities/Author.ts

```
18 @Entity({ name: "Authors" })
19 @ObjectType()
20 export class Author {
21   @PrimaryGeneratedColumn()
22   @Field()
23   id!: number;
24
25   @Column()
26   @Field()
27   name!: string;
```

The new decorators are: - `@Entity` that declares the type to be a entity in our database; thats equates to a table in sql speak - `@Column` that declares that field part of our

database entry - `@PrimaryGeneratedColumn()` denotes it's a special column that acts as a PrimaryKey and is auto generated

All of those decorators take additional options to further refine exact realization.

We then repeat the same with our Post class:

code/9-typegraphql/with-typeorm/entities/Post.ts

```
21 @Entity({ name: "Posts" })
22 @ObjectType()
23 export class Post {
24   @PrimaryGeneratedColumn()
25   @Field()
26   id!: number;
27
28   @Column()
29   @Field()
30   title!: string;
31
32   @Column()
33   @Field()
34   text!: string;
35
36   @CreateDateColumn()
37   createdAt!: Date;
38
39   @UpdateDateColumn()
40   updatedAt!: Date;
41
42   @Column()
43   authorId!: number;
44 }
```

Apart of the usual we here added even more special decorators: - `@CreateDateColumn()` which adds a field that automatically gets populated with the current timestamp on creation - `@UpdateDateColumn()` which adds a field that automatically gets populated with the current timestamp on update

Also note that authorId is declared as a `@Column()` but is not a field. and hence is not exposed in the GraphQL schema.

Next we need to bootstrap our database and update our `main()`:

code/9-typegraphql/with-typeorm/index.ts

```
15 async function main() {
16   // build TypeGraphQL executable schema
17   const [schema, connection] = await Promise.all([
18     buildSchema({ resolvers: [AuthorResolver, PostResolver] }),
19     createConnection(options),
20   ]);
21
22   connection.synchronize();
23
24   // setup context
25   const context = {
26     authors: connection.getRepository(Author),
27     posts: connection.getRepository(Post),
28   };
29
30   // Create GraphQL server
31   const server = new ApolloServer({ schema, context: () => context });
32
33   // Start the server
34   const { url } = await server.listen(4000);
35   console.log(`Server is running, GraphQL Playground available at ${url\
36 }`);
37 }
```

This initializes the TypeORM a sqlite connection and passes a list of entities to TypeORM.

Then we call `synchronize` which creates our tables in our database in case they don't exist yet. Lastly we pass objects that represent our tables to our context.

> Of course, you can use any database you'd like. We use PostgreSQL at newline, but we use sqlite here because it's portable and easy to get started.

Configuring Context

The type of context is now:

code/9-typegraphql/with-typeorm/context.ts

```
5  export interface Context {
6    authors: Repository<Author>;
7    posts: Repository<Post>;
8  }
```

and consequently our resolvers need to change.

We can retrieve authors via `findAll`:

code/9-typegraphql/with-typeorm/entities/Author.1.ts

```
30 @Resolver(() => Author)
31 export class AuthorResolver {
32   @Query(() => [Author])
33   async authors(
34     @Ctx() context: Context,
35     @Args()
36     { offset, limit }: PaginationInput
37   ): Promise<Author[]> {
38     const authors = await context.authors.findAll({
39       skip: offset,
40       take: limit,
41     });
42
43     return authors;
44   }
```

`findAll` returns a list of `authors` and a `count`.

The new `addAuthors` mutation will then use `.save`:

code/9-typegraphql/with-typeorm/entities/Author.1.ts

```
60   @Mutation(() => Author)
61   async addAuthor(@Ctx() context: Context, @Arg("name") name: string) {
62     const author = context.authors.create();
63     author.name = name;
64     return context.authors.save(author);
65   }
```

`.save(author)` returns a promise of the newly inserted object.

Navigating Relationships

One of the great things about GraphQL is that we can traverse relationships. For example, we can find the `Posts` that were written by an `Author`.

Here's one way to do this:

code/9-typegraphql/with-typeorm/entities/Author.1.ts

```
46   @FieldResolver(() => [Post])
47   async posts(
48     @Ctx() context: Context,
49     @Root() author: Author,
50     @Args() { offset, limit }: PaginationInput
51   ): Promise<Post[]> {
52     const posts = await context.posts.findAll({
53       where: { authorId: author.id },
54       skip: offset,
55       take: limit,
56     });
57     return posts;
58   }
```

Note the `where` clause. This is a simple and intuitive API but if this API is not sufficient we could also use the query builder API.

However, in our case we can actually get rid of the resolver entirely and let TypeORM do the resolution.

OneToMany with TypeORM

To do this, we need to define the relation between authorId and Author. We can do this by using `@OneToMany`:

code/9-typegraphql/with-typeorm/entities/Author.ts

```
30   @OneToMany(() => Post, (post: Post) => post.authorId, { lazy: true })
31   @Field(() => [Post])
32   posts!: Promise<Post>
```

The 3 parameters of `OneToMany` have the following meaning:

1. specifies which type we will be returning
2. defines which column we will be using for the join operation
3. denotes that this join will only be performed if that field is accessed. This is crucial as we don't want to fetch more data then necessary.

That provides a nice tool set to quickly build out an application - but there's a lot more features TypeORM has to offer.

Next, instead of getting lost in the depths of TypeORM less bounce back to another set of TypeGraphQL decorators.

Let's have a look how we can solve authorization in TypeGraphQL:

Authorization with TypeGraphQL

As we already covered authorization in depth in the previous chapter, we will not go into detail how to implement a proper authorization solution and instead just highlight what is provided in this framework.

TypeGraphQL has support for an `authChecker` method and provides a dedicated set of directives for declaring what fields require authentication.

The library provides a role based authentication system with field granularity. This works by decorating fields with `@Authenticated(role)` to attach a accessibility policy to a field. The actual logic of authentication and checking then needs to be implemented in the `authChecker`. So lets see how this looks in code:

code/9-typegraphql/with-authorization/entities/Post.ts

```
43   @Authorized("ADMIN")
44   @Field()
45   view!: number;
```

Then one needs to pass an `authChecker` to the buildSchema:

code/9-typegraphql/with-authorization/index.ts

```
29 async function main() {
30   // build TypeGraphQL executable schema
31   const [schema, connection] = await Promise.all([
32     buildSchema({
33       authChecker,
34       resolvers: [AuthorResolver, PostResolver],
35     }),
36     createConnection(options),
37   ]);
```

The `authChecker` could be implemented along those lines:

code/9-typegraphql/with-authorization/index.ts

```
21 function authChecker({ context }: ResolverData<Context>, roles: string[\
22 ]) {
23   if (roles.length === 0) {
24     return hasRole(context, "");
25   } else {
26     return roles.filter((role) => hasRole(context, role)).length > 0;
27   }
28 }
```

Where `hasRole` is your custom logic that determine if the user has the specified role assigned.

And that's it!

Summary

This chapter showed how you create a code first GraphQL server using `type-graphql` that no longer requires you to write a GraphQL schema by hand but can generate it using decorators. This harnesses the full power of TypeScript, as your server is already type-safe.

By using the library TypeORM you can query databases with GraphqL using the database schema. Also, authorization can be added to a code first server. Remember:

- You only need to add decorators for public facing fields and you don't need to expose methods that are only for implementation purposes
- use context to reduce keep the state of your Resolver Classes minimal. This makes things easier to test.
- to leverage TypeORMs One-To-Many decorators to model the relationship between your entities

More on TypeGraphQL

Besides decorators for cost computation and authorization, you also can find many utility decorators written by the community. One great package is for instance the `class-validation` package which offers decorators for validating properties of individual arguments.

TypeGraphQL and TypeORM let you build apps very quickly and allows you to keep everything in one central place. However, both require TypeScript with meta-data reflection which is not fully supported by babel yet.

As this approach relies on runtime reflection it has also a higher overhead to other frameworks. TypeGraphQL+TypeORM is not the only Code-First framework but the only one that embraces Typescript decorators. Another recommendable code-first framework with similar principles and many integrations is NexusJS[35]

[35] https://nexusjs.org/

Where to Go From Here? Advanced GraphQL

You've made it to the end! I want to give you a few guidelines on where you can go from here.

What did you think?

First off, what did you think of this book? Did you find it helpful?

We would super-appreciate it if you would click here[36] and let us know what you thought of the book.

If you want to request any new chapters click here[37] and let us know!

Awesome GraphQL

You can check out a comprehensive list of GraphQL resources in the awesome-graphql[38] repo.

There you'll find libraries for pretty much every language, example code, tools, tutorials and more.

Say Hello

We talk about GraphQL on Twitter and you can find us on Twitter here. Hop on and say hello!

[36] https://newline.co/l/fullstack-graphql-review
[37] https://newline.co/l/request-content
[38] https://github.com/chentsulin/awesome-graphql

- Gaetano[39]
- Roy[40]
- Nate Murray[41]

[39] https://twitter.com/tanoChecinski
[40] https://twitter.com/gethackteam
[41] https://twitter.com/fullstackio