Contents

[GraphQL 7](#_Toc145423658)

[Introduction 7](#_Toc145423659)

[GraphQL - A Query Language for APIs 7](#_Toc145423660)

[A more efficient Alternative to REST 7](#_Toc145423661)

[History, Context & Adoption 8](#_Toc145423662)

[GraphQL is *not* only for React Developers 8](#_Toc145423663)

[A rapidly growing Community 8](#_Toc145423664)

[GraphQL is the Better REST 10](#_Toc145423665)

[REST API Drawback #1: Over fetching 10](#_Toc145423666)

[REST API Drawback #2: Under fetching 10](#_Toc145423667)

[Single Endpoint 11](#_Toc145423668)

[Data Fetching with REST vs GraphQL 12](#_Toc145423669)

[No more Over- and Underfetching 14](#_Toc145423670)

[Overfetching: Downloading superfluous data 14](#_Toc145423671)

[Underfetching and the n+1 problem 14](#_Toc145423672)

[Rapid Product Iterations on the Frontend 14](#_Toc145423673)

[Insightful Analytics on the Backend 15](#_Toc145423674)

[Benefits of a Schema & Type System 15](#_Toc145423675)

[GraphQL – Core Concepts 16](#_Toc145423676)

[The Schema Definition Language (SDL) 16](#_Toc145423677)

[Fetching Data with Queries 17](#_Toc145423678)

[Basic Queries 17](#_Toc145423679)

[Walking Through a Graph 18](#_Toc145423680)

[Queries with Arguments 21](#_Toc145423681)

[Writing Data with Mutations 21](#_Toc145423682)

[Realtime Updates with Subscriptions 22](#_Toc145423683)

[Defining a Schema 23](#_Toc145423684)

[GraphQL – The Big Picture – Architecture 26](#_Toc145423685)

[Use Cases 26](#_Toc145423686)

[1. GraphQL server with a connected database 26](#_Toc145423687)

[2. GraphQL layer that integrates existing systems 27](#_Toc145423688)

[3. Hybrid approach with connected database and integration of existing system 27](#_Toc145423689)

[Resolver Functions 28](#_Toc145423690)

[GraphQL Client Libraries 29](#_Toc145423691)

[GraphQL – Advanced Topic – Clients 30](#_Toc145423692)

[Send Queries and Mutations Directly 30](#_Toc145423693)

[View Layer Integrations & UI updates 30](#_Toc145423694)

[Caching Query Results: Concepts and Strategies 31](#_Toc145423695)

[Build-time Schema Validation & Optimizations 31](#_Toc145423696)

[Colocating Views and Data Dependencies 31](#_Toc145423697)

[GraphQL – Advanced Topic – Server 32](#_Toc145423698)

[GraphQL execution 32](#_Toc145423699)

[Batched Resolving 33](#_Toc145423700)

[GraphQL – More Concepts 35](#_Toc145423701)

[Enhancing Reusability with Fragments 35](#_Toc145423702)

[Parameterizing Fields with Arguments 35](#_Toc145423703)

[Named Query Results with Aliases 36](#_Toc145423704)

[Advanced SDL 37](#_Toc145423705)

[Object & Scalar Types 37](#_Toc145423706)

[Enums 37](#_Toc145423707)

[Interface 37](#_Toc145423708)

[Union Types 38](#_Toc145423709)

[Tooling and Ecosystem 39](#_Toc145423710)

[Introspection 39](#_Toc145423711)

[GraphQL Playground 41](#_Toc145423712)

[GraphQL – Security 42](#_Toc145423713)

[Timeout 42](#_Toc145423714)

[Timeout Pros 42](#_Toc145423715)

[Timeout Cons 42](#_Toc145423716)

[Maximum Query Depth 42](#_Toc145423717)

[Maximum Query Depth Pros 43](#_Toc145423718)

[Maximum Query Depth Cons 44](#_Toc145423719)

[Query Complexity 44](#_Toc145423720)

[Query Complexity Pros 44](#_Toc145423721)

[Query Complexity Cons 44](#_Toc145423722)

[Throttling 45](#_Toc145423723)

[Throttling Based on Server Time 45](#_Toc145423724)

[Throttling Based on Query Complexity 46](#_Toc145423725)

[Summary 46](#_Toc145423726)

[GraphQL – Common Questions 47](#_Toc145423727)

[Is GraphQL a Database Technology? 47](#_Toc145423728)

[Is GraphQL only for React / Javascript Developers? 47](#_Toc145423729)

[How to do Server-side Caching? 47](#_Toc145423730)

[How to do Authentication and Authorization? 47](#_Toc145423731)

[How to do Error Handling? 48](#_Toc145423732)

[Does GraphQL Support Offline Usage? 48](#_Toc145423733)

[React + Apollo Tutorial – Introduction 49](#_Toc145423734)

[Overview 49](#_Toc145423735)

[Why a GraphQL Client? 49](#_Toc145423736)

[Apollo vs Relay vs urql 50](#_Toc145423737)

[Relay 50](#_Toc145423738)

[Apollo 50](#_Toc145423739)

[urql 50](#_Toc145423740)

[GraphQL Example with Node.JS and Apollo 51](#_Toc145423741)

[Setup Apollo Server 51](#_Toc145423742)

[Step 1: Create a new project 51](#_Toc145423743)

[Step 2: Install dependencies 51](#_Toc145423744)

[Step 3: Define your GraphQL schema 52](#_Toc145423745)

[Step 4: Define your data set 54](#_Toc145423746)

[Step 5: Define a resolver 54](#_Toc145423747)

[Step 6: Create an instance of ApolloServer 54](#_Toc145423748)

[Step 7: Start the server 55](#_Toc145423749)

[Step 8: Execute your first query 55](#_Toc145423750)

[Game, Review and Author 56](#_Toc145423751)

[Configure the Data 59](#_Toc145423752)

[Resolver Functions 59](#_Toc145423753)

[Query Variables 61](#_Toc145423754)

[Related Data 63](#_Toc145423755)

[Mutations – Adding & Deleting Data 68](#_Toc145423756)

[Mutations – Updating Data 71](#_Toc145423757)

[Calling the GraphQL API Using curl 73](#_Toc145423758)

[Calling the GraphQL API Using Postman 73](#_Toc145423759)

[Calling the GraphQL API Using fetch() 73](#_Toc145423760)

[React + Apollo – Getting Started 75](#_Toc145423761)

[Frontend 75](#_Toc145423762)

[Prepare Styling 76](#_Toc145423763)

[Install Apollo Client 78](#_Toc145423764)

[Configure ApolloClient 78](#_Toc145423765)

[Backend 80](#_Toc145423766)

[Downloading the Server Code 80](#_Toc145423767)

[Creating a Database and Generating Prisma Client 83](#_Toc145423768)

[Exploring the server 83](#_Toc145423769)

[React + Apollo – Queries: Loading Links 87](#_Toc145423770)

[Preparing the React components 87](#_Toc145423771)

[Writing the GraphQL query 89](#_Toc145423772)

[Queries with Apollo Client 89](#_Toc145423773)

[React + Apollo – Mutations: Creating Links 91](#_Toc145423774)

[Preparing the React components 91](#_Toc145423775)

[Writing the mutation 92](#_Toc145423776)

[React + Apollo – Routing 96](#_Toc145423777)

[Create a Header 96](#_Toc145423778)

[Setup routes 97](#_Toc145423779)

[Implement Navigation 98](#_Toc145423780)

[React + Apollo – Authentication 100](#_Toc145423781)

[Prepare the React Components 100](#_Toc145423782)

[Using the authentication mutations 104](#_Toc145423783)

[Configuring Apollo with the authentication token 107](#_Toc145423784)

[Requiring authentication on the server-side 108](#_Toc145423785)

[React + Apollo – More Mutations and Updating the Store 110](#_Toc145423786)

[Preparing the React Components 110](#_Toc145423787)

[Calling the Mutation 114](#_Toc145423788)

[Updating the cache 115](#_Toc145423789)

[React + Apollo – Filtering: Searching the List of Links 119](#_Toc145423790)

[Preparing the React components 119](#_Toc145423791)

[Filtering Links 121](#_Toc145423792)

[React + Apollo – Realtime Updates with GraphQL Subscriptions 124](#_Toc145423793)

[What are GraphQL Subscriptions? 124](#_Toc145423794)

[Subscriptions with Apollo 124](#_Toc145423795)

[Subscribing to New links 126](#_Toc145423796)

[Subscribing to New Votes 128](#_Toc145423797)

[React + Apollo – Pagination 130](#_Toc145423798)

[Preparing the React Components 130](#_Toc145423799)

[Implementing Navigation 133](#_Toc145423800)

[Final Adjustments 136](#_Toc145423801)

[GraphQL.NET 138](#_Toc145423802)

[Introduction 138](#_Toc145423803)

[Installation 138](#_Toc145423804)

[GraphiQL – IDE 138](#_Toc145423805)

[Altair GraphQL Client 138](#_Toc145423806)

[Queries 138](#_Toc145423807)

[Schema Types 138](#_Toc145423808)

[Custom Scalars 139](#_Toc145423809)

[Lists and Non-Null 139](#_Toc145423810)

[Arguments 139](#_Toc145423811)

[Aliases 139](#_Toc145423812)

[Fragments 139](#_Toc145423813)

[Variables 139](#_Toc145423814)

[Directives 139](#_Toc145423815)

[Mutations 139](#_Toc145423816)

[Interfaces 139](#_Toc145423817)

[Unions 139](#_Toc145423818)

[Subscriptions 140](#_Toc145423819)

[Query Validation 140](#_Toc145423820)

[Query Organization 140](#_Toc145423821)

[User Context 140](#_Toc145423822)

[Error Handling 140](#_Toc145423823)

[Dependency Injection 140](#_Toc145423824)

[How do I use XYZ ORM/database with GraphQL.NET? 140](#_Toc145423825)

[Protection Against Malicious Queries 140](#_Toc145423826)

[Object/Field Metadata 140](#_Toc145423827)

[Field Middleware 140](#_Toc145423828)

[Metrics 141](#_Toc145423829)

[Authorization 141](#_Toc145423830)

[Global Switches 141](#_Toc145423831)

[Intro to GraphQL for .NET Developers: Schema, Resolver, and Query Language 142](#_Toc145423832)

[Implementing a TechEvent Management GraphQL with ASP.Net Core 143](#_Toc145423833)

[.Net 5 API with GraphQL - Step by Step Using Hot Chocolate 144](#_Toc145423834)

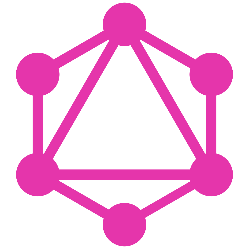
[How to implement GraphQL in ASP.Net Core Web API (.NET 6) using HotChocolate 145](#_Toc145423835)

# GraphQL

<https://graphql.org/learn/>

<https://www.howtographql.com/>

<https://www.youtube.com/watch?v=5199E50O7SI>



## Introduction

<https://www.howtographql.com/basics/0-introduction/>

* GraphQL is a Query Language.
* It is an alternative to using a REST API.

[GraphQL](http://www.graphql.org/) is a new API standard that provides a more efficient, powerful, and flexible alternative to REST. It was developed and [open-sourced by Facebook](https://facebook.github.io/react/blog/2015/02/20/introducing-relay-and-graphql.html) and is now maintained by a large community of companies and individuals from all over the world.

APIs have become ubiquitous components of software infrastructures. In short, an **API** defines how a **client** can load data from a **server**.

At its core, GraphQL enables declarative data fetching where a client can specify exactly what data it needs from an API. Instead of multiple endpoints that return fixed data structures, a GraphQL server only exposes a single endpoint and responds with precisely the data a client asked for.

### GraphQL - A Query Language for APIs

Most applications today have the need to fetch data from a server where that data is stored in a database. It’s the responsibility of the API to provide an interface to the stored data that fits an application’s needs.

A black and white cloud with arrows

Description automatically generated with medium confidence

GraphQL is often confused with being a database technology. This is a misconception, GraphQL is a query language for APIs - not databases. In that sense it’s database agnostic and effectively can be used in any context where an API is used.

### A more efficient Alternative to REST

💡 Learn more about the top reasons to use GraphQL in [this](https://www.prisma.io/blog/top-5-reasons-to-use-graphql-b60cfa683511) blog post.

[REST](https://en.wikipedia.org/wiki/Representational_state_transfer) has been a popular way to expose data from a server. When the concept of REST was developed, client applications were relatively simple, and the development pace wasn’t nearly where it is today. REST thus was a good fit for many applications. However, the API landscape has radically changed over the last couple of years. In particular, there are three factors that have been challenging the way APIs are designed:

1. Increased mobile usage creates need for efficient data loading.

Increased mobile usage, low-powered devices and sloppy networks were the initial reasons why Facebook developed GraphQL. GraphQL minimizes the amount of data that needs to be transferred over the network and thus majorly improves applications operating under these conditions.

1. Variety of different frontend frameworks and platforms

The heterogeneous landscape of frontend frameworks and platforms that run client applications makes it difficult to build and maintain one API that would fit the requirements of all. With GraphQL, each client can access precisely the data it needs.

1. Fast development & expectation for rapid feature development

Continuous deployment has become a standard for many companies, rapid iterations and frequent product updates are indispensable. With REST APIs, the way data is exposed by the server often needs to be modified to account for specific requirements and design changes on the client-side. This hinders fast development practices and product iterations.

## History, Context & Adoption

### GraphQL is not only for React Developers

Facebook started using GraphQL in 2012 in their native mobile apps. Interestingly though, GraphQL has mostly been picked up to be used in the context of web technologies and has gained only little traction in the native mobile space.

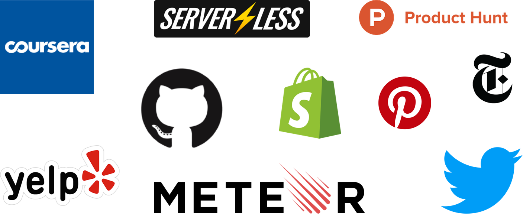
The first time Facebook publicly spoke about GraphQL was at [React.js Conf 2015](https://www.youtube.com/watch?v=9sc8Pyc51uU) and shortly after announced their [plans to open source](https://facebook.github.io/react/blog/2015/05/01/graphql-introduction.html) it. Because Facebook always used to speak about GraphQL in the context of [React](https://facebook.github.io/react/), it took a while for non-React developers to understand that GraphQL was by no means a technology that was limited to usage with React.

[Dan Schafer](https://twitter.com/dlschafer) & [Jing Chen](https://twitter.com/jingc?lang=en) publicly introduced GraphQL at React.JS Conf 2015. [Watch video](https://www.youtube.com/watch?v=9sc8Pyc51uU).

### A rapidly growing Community

In fact, GraphQL is a technology that can be used everywhere a client communicates with an API. Interestingly, other companies like [Netflix](https://medium.com/netflix-techblog) or [Coursera](https://building.coursera.org/) were working on comparable ideas to make API interactions more efficient. Coursera envisioned a similar technology to let a client specify its data requirements and Netflix even open-sourced their solution called [Falcor](https://github.com/Netflix/falcor). After GraphQL was open-sourced, Coursera completely cancelled their own efforts and hopped on the GraphQL train.

Today, GraphQL is used in production by [lots of different companies](http://graphql.org/users/) such as GitHub, Twitter, Yelp and Shopify - to name only a few.



Despite being such a young technology, GraphQL has already been widely adopted. Learn [who else](http://graphql.org/users/) is using GraphQL in production.

There are entire conferences dedicated to GraphQL such as [GraphQL Conf](https://graphqlconf.org/) and more resources like the [GraphQL Weekly](https://graphqlweekly.com/) newsletter.

# GraphQL is the Better REST

<https://www.howtographql.com/basics/1-graphql-is-the-better-rest/>

Over the past decade, [REST](https://en.wikipedia.org/wiki/Representational_state_transfer) has become the standard (yet a fuzzy one) for designing web APIs. It offers some great ideas, such as stateless servers and structured access to resources. However, REST APIs have shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them.

GraphQL was developed to cope with the need for more flexibility and efficiency! It solves many of the shortcomings and inefficiencies that developers experience when interacting with REST APIs.

## REST API Drawback #1: Over fetching

Getting back more data than we need.

mysite.com/api/course:

{

"id": "1",

"title": "Thud",

"author": {...},

"price": "10.00",

"thumbnail\_url": "...",

"video\_url": "...",

}

But we need only the id, title and thumbnail\_url for each course to show on a page. So, the rest of the data is not required. Which means, we over fetched.

## REST API Drawback #2: Under fetching

Getting back less data than we need.

We may have to make multiple requests to different endpoints to collect everything we need together. For e.g.;

mysite.com/api/course/1:

{

"id": "1",

"title": "Thud",

"author": {...},

"price": "10.00",

"thumbnail\_url": "...",

"video\_url": "...",

}

Additional request to fetch the additional author data:

mysite.com/api/author/1:

These issues can be easily solved by using GraphQL instead.

To illustrate the major differences between REST and GraphQL when it comes to fetching data from an API, let’s consider a simple example scenario: In a blogging application, an app needs to display the titles of the posts of a specific user. The same screen also displays the names of the last 3 followers of that user. How would that situation be solved with REST and GraphQL?

💡 Check out [this](https://www.prisma.io/blog/top-5-reasons-to-use-graphql-b60cfa683511) article to learn more about why developers love GraphQL.

## Single Endpoint

mygraphqlsite.com/graphql

Different from a REST API where each resource typically has its own set of endpoints for GET POST PUT DELETE request etc. Whenever we send a request to a GraphQL server, it is always going to be sent to probably that single endpoint and then the server can handle it.

The we send a query to the server is by using a special GraphQL syntax that looks something like this:

query {

courses {

id

name

thumbnail\_url

}

}

This syntax allows us to specify what data and what fields we need back from the server. So, in the example from before, if we want to fetch courses data, we can send a query that looks like the one shown above.

GraphQL allows us to fetch nested related data within a single query. So again, from the example before where we need a single course, we’d send a query like this one:

query {

course(id: “1”) {

id

name

thumbnail\_url

author {

id

name

courses {

id

title

thumbnail\_url

}

}

}

}

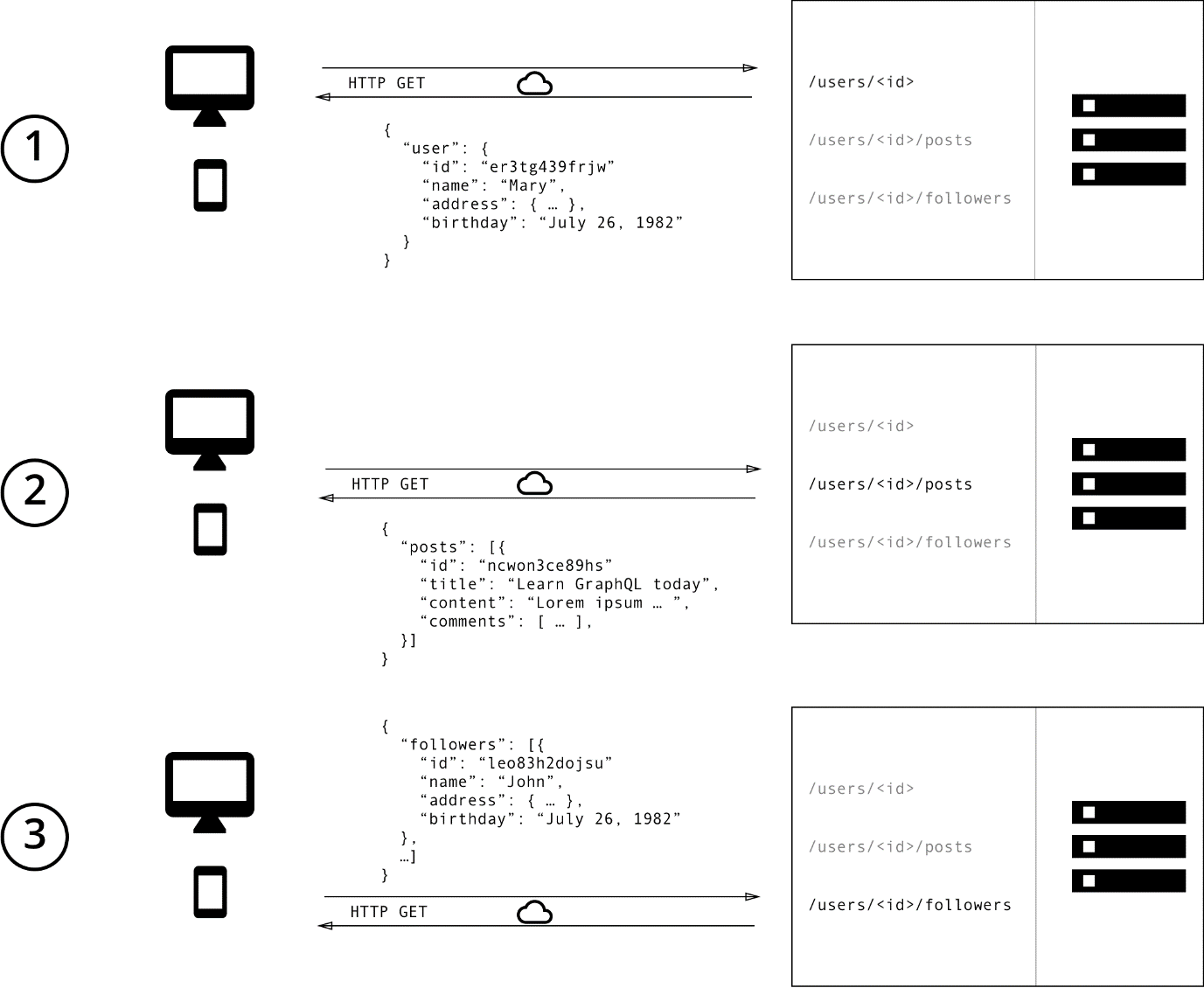
Here, we also specified that we wanted extra information about the author of that course along with any related data or any related courses, rather, that the author made. We do that in GraphQL by nesting those properties inside the query. All this will be a single request (or query). No longer under fetching the data that we need.

We can also perform **mutations**, to ask the GraphQL server to add new data or update it or delete it.

## Data Fetching with REST vs GraphQL

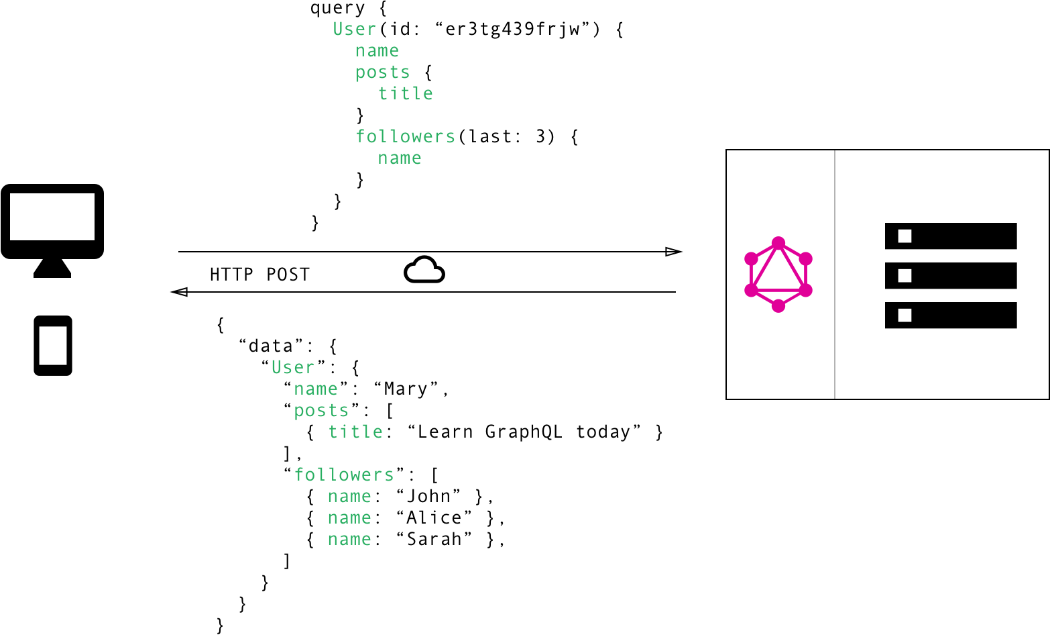
With a REST API, you would typically gather the data by accessing multiple endpoints. In the example, these could be /users/<id> endpoint to fetch the initial user data. Secondly, there’s likely to be a /users/<id>/posts endpoint that returns all the posts for a user. The third endpoint will then be the /users/<id>/followers that returns a list of followers per user.

|  |
| --- |
| A screenshot of a phone  Description automatically generated |
| A screen shot of a computer  Description automatically generated |



With REST, you have to make three requests to different endpoints to fetch the required data. You’re also overfetching since the endpoints return additional information that’s not needed.

In GraphQL on the other hand, you’d simply send a single query to the GraphQL server that includes the concrete data requirements. The server then responds with a JSON object where these requirements are fulfilled.



Using GraphQL, the client can specify exactly the data it needs in a query. Notice that the structure of the server’s response follows precisely the nested structure defined in the query.

## No more Over- and Underfetching

One of the most common problems with REST is that of over- and underfetching. This happens because the only way for a client to download data is by hitting endpoints that return fixed data structures. It’s very difficult to design the API in a way that it’s able to provide clients with their exact data needs.

“Think in graphs, not endpoints.” [Lessons From 4 Years of GraphQL](http://www.graphql.com/articles/4-years-of-graphql-lee-byron) by [Lee Byron](https://twitter.com/leeb), GraphQL Co-Inventor.

### Overfetching: Downloading superfluous data

Overfetching means that a client downloads more information than is actually required in the app. Imagine for example a screen that needs to display a list of users only with their names. In a REST API, this app would usually hit the /users endpoint and receive a JSON array with user data. This response however might contain more info about the users that are returned, e.g. their birthdays or addresses - information that is useless for the client because it only needs to display the users’ names.

### Underfetching and the n+1 problem

Another issue is underfetching and the n+1-requests problem. Underfetching generally means that a specific endpoint doesn’t provide enough of the required information. The client will have to make additional requests to fetch everything it needs. This can escalate to a situation where a client needs to first download a list of elements, but then needs to make one additional request per element to fetch the required data.

As an example, consider the same app would also need to display the last three followers per user. The API provides the additional endpoint /users/<user-id>/followers. In order to be able to display the required information, the app will have to make one request to the /users endpoint and then hit the /users/<user-id>/followers endpoint for each user.

## Rapid Product Iterations on the Frontend

A common pattern with REST APIs is to structure the endpoints according to the views that you have inside your app. This is handy since it allows for the client to get all required information for a particular view by simply accessing the corresponding endpoint.

The major drawback of this approach is that it doesn’t allow for rapid iterations on the frontend. With every change that is made to the UI, there is a high risk that now there is more (or less) data required than before. Consequently, the backend needs to be adjusted as well to account for the new data needs. This kills productivity and notably slows down the ability to incorporate user feedback into a product.

With GraphQL, this problem is solved. Thanks to the flexible nature of GraphQL, changes on the client-side can be made without any extra work on the server. Since clients can specify their exact data requirements, no backend engineer needs to make adjustments when the design and data needs on the frontend change.

## Insightful Analytics on the Backend

GraphQL allows you to have fine-grained insights about the data that’s requested on the backend. As each client specifies exactly what information it’s interested in, it is possible to gain a deep understanding of how the available data is being used. This can for example help in evolving an API and deprecating specific fields that are not requested by any clients any more.

With GraphQL, you can also do low-level performance monitoring of the requests that are processed by your server. GraphQL uses the concept of resolver functions to collect the data that’s requested by a client. Instrumenting and measuring performance of these resolvers provides crucial insights about bottlenecks in your system.

## Benefits of a Schema & Type System

GraphQL uses a strong type system to define the capabilities of an API. All the types that are exposed in an API are written down in a schema using the GraphQL Schema Definition Language (SDL). This schema serves as the contract between the client and the server to define how a client can access the data.

Once the schema is defined, the teams working on frontend and backends can do their work without further communication since they both are aware of the definite structure of the data that’s sent over the network.

Frontend teams can easily test their applications by mocking the required data structures. Once the server is ready, the switch can be flipped for the client apps to load the data from the actual API.

# GraphQL – Core Concepts

<https://www.howtographql.com/basics/2-core-concepts/>

In this chapter, you’ll learn about some fundamental language constructs of GraphQL. That includes a first glimpse at the syntax for defining types as well as sending queries and mutations.

## The Schema Definition Language (SDL)

GraphQL has its own type system that’s used to define the schema of an API. The syntax for writing schemas is called [Schema Definition Language](https://www.prisma.io/blog/graphql-sdl-schema-definition-language-6755bcb9ce51) (SDL).

Here is an example of how we can use the SDL to define a simple type called Person:

type Person {

name: String!

age: Int!

}

This type has two fields, they’re called name and age and are respectively of type String and Int. The ! following the type means that this field is required.

A computer code with text

Description automatically generated with medium confidence

It’s also possible to express relationships between types. In the example of a blogging application, a Person could be associated with a Post:

type Post {

title: String!

author: Person!

}

Conversely, the other end of the relationship needs to be placed on the Person type:

type Person {

name: String!

age: Int!

posts: [Post!]!

}

Note that we just created a one-to-many-relationship between Person and Post since the posts field on Person is actually an array of posts.

A screenshot of a computer

Description automatically generated

## Fetching Data with Queries

When working with REST APIs, data is loaded from specific endpoints. Each endpoint has a clearly defined structure of the information that it returns. This means that the data requirements of a client are effectively encoded in the URL that it connects to.

The approach that’s taken in GraphQL is radically different. Instead of having multiple endpoints that return fixed data structures, GraphQL APIs typically only expose a single endpoint. This works because the structure of the data that’s returned is not fixed. Instead, it’s completely flexible and lets the client decide what data is actually needed.

That means that the client needs to send more information to the server to express its data needs - this information is called a query.

**Note**: Unfortunately, we no longer provide the Run in Sandbox feature that is demonstrated in the video at 13:50. We are really sorry for the inconvenience.

### Basic Queries

Let’s take a look at an example query that a client could send to a server:

{

allPersons {

name

}

}

The allPersons field in this query is called the root field of the query. Everything that follows the root field, is called the payload of the query. The only field that’s specified in this query’s payload is name.

This query would return a list of all persons currently stored in the database. Here’s an example response:

{

"allPersons": [

{ "name": "Johnny" },

{ "name": "Sarah" },

{ "name": "Alice" }

]

}

Notice that each person only has the name in the response, but the age is not returned by the server. That’s exactly because name was the only field that was specified in the query.

If the client also needed the persons’ age, all it has to do is slightly adjust the query and include the new field in the query’s payload:

|  |  |
| --- | --- |
| {  allPersons {  name  age  }  } | A close-up of a text  Description automatically generated |

One of the major advantages of GraphQL is that it allows for naturally querying nested information. For example, if you wanted to load all the posts that a Person has written, you could simply follow the structure of your types to request this information:

|  |  |
| --- | --- |
| {  allPersons {  name  age  posts {  title  }  }  } |  |

### Walking Through a Graph

<https://www.youtube.com/watch?v=5199E50O7SI>

Let’s see how we query a graph and move around it to navigate data.

When we make a GraphQL server or API, we’re making something called a **graph**. In visual terms, a graph is basically a bunch of connected data that looks something like this:

A blue foot prints with white text

Description automatically generated

So, in this case, we have 3 different data types, Games, Authors and Reviews. GraphQL allows us to traverse or walk thru this graph to also fetch any related data to that starting point. So, the starting or landing point may be *reviews*:

query {

reviews {

rating

}

}

From there, I can say also get me the *author* of each review that I got back and I could also specify which fields of the *authors* that I want to get back as well.

query {

reviews {

rating,

author {

name

}

}

}

A diagram of a foot

Description automatically generated with medium confidence

The reason I could do this is because when I made the GraphQL server, I connected these data resources. I said that each review was related to an author who wrote that review, and the author is a separate resource. All this data would be brought back from a single request. We’ve only made one query. We didn’t have to first get the reviews and then make a second request for the authors for each of those review, even though it’s a separate resource that we’re getting right here.

Another example could be, my initial entry point to the graph would be a specific game with a certain ID and the query for that would look something like this:

query {

game(id: “2”) {

title

}

}

Now, I could also say, get me any review related to that game, and from those reviews just get me the rating field. And to take it one step further, I could also say then get the author for each of those reviews and just give me their names.

query {

game(id: “2”) {

title,

review {

rating,

author {

name

}

}

}

}

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

### Queries with Arguments

In GraphQL, each field can have zero or more arguments if that’s specified in the schema. For example, the allPersons field could have a last parameter to only return up to a specific number of persons. Here’s what a corresponding query would look like:

{

allPersons(last: 2) {

name

}

}

## Writing Data with Mutations

Next to requesting information from a server, the majority of applications also need some way of making changes to the data that’s currently stored in the backend. With GraphQL, these changes are made using so-called mutations. There generally are three kinds of mutations:

* creating new data
* updating existing data
* deleting existing data

Mutations follow the same syntactical structure as queries, but they always need to start with the mutation keyword. Here’s an example for how we might create a new Person:

mutation {

createPerson(name: "Bob", age: 36) {

name

age

}

}

Notice that similar to the query we wrote before, the mutation also has a root field - in this case it’s called createPerson. We also already learned about the concepts of arguments for fields. In this case, the createPerson field takes two arguments that specify the new person’s name and age.

Like with a query, we’re also able to specify a payload for a mutation in which we can ask for different properties of the new Person object. In our case, we’re asking for the name and the age - though admittedly that’s not super helpful in our example since we obviously already know them as we pass them into the mutation. However, being able to also query information when sending mutations can be a very powerful tool that allows you to retrieve new information from the server in a single roundtrip!

The server response for the above mutation would look as follows:

"createPerson": {

"name": "Bob",

"age": 36,

}

One pattern you’ll often find is that GraphQL types have unique IDs that are generated by the server when new objects are created. Extending our Person type from before, we could add an id like this:

type Person {

id: ID!

name: String!

age: Int!

}

Now, when a new Person is created, you could directly ask for the id in the payload of the mutation, since that is information that wasn’t available on the client beforehand:

mutation {

createPerson(name: "Alice", age: 36) {

id

}

}

## Realtime Updates with Subscriptions

Another important requirement for many applications today is to have a realtime connection to the server in order to get immediately informed about important events. For this use case, GraphQL offers the concept of subscriptions.

When a client subscribes to an event, it will initiate and hold a steady connection to the server. Whenever that particular event then actually happens, the server pushes the corresponding data to the client. Unlike queries and mutations that follow a typical “request-response-cycle”, subscriptions represent a stream of data sent over to the client.

Subscriptions are written using the same syntax as queries and mutations. Here’s an example where we subscribe on events happening on the Person type:

subscription {

newPerson {

name

age

}

}

After a client sent this subscription to a server, a connection is opened between them. Then, whenever a new mutation is performed that creates a new Person, the server sends the information about this person over to the client:

{

"newPerson": {

"name": "Jane",

"age": 23

}

}

## Defining a Schema

Now that you have a basic understanding of what queries, mutations, and subscriptions look like, let’s put it all together and learn how you can write a schema that would allow you to execute the examples you’ve seen so far.

The schema is one of the most important concepts when working with a GraphQL API. It specifies the capabilities of the API and defines how clients can request the data. It is often seen as a contract between the server and client.

A computer screen shot of a computer code

Description automatically generated

Generally, a schema is simply a collection of GraphQL types. However, when writing the schema for an API, there are some special root types:

type Query { ... }

type Mutation { ... }

type Subscription { ... }

The Query, Mutation, and Subscription types are the entry points for the requests sent by the client. To enable the allPersons-query that we saw before, the Query type would have to be written as follows:

type Query {

allPersons: [Person!]!

}

allPersons is called a root field of the API. Considering again the example where we added the last argument to the allPersons field, we’d have to write the Query as follows:

type Query {

allPersons(last: Int): [Person!]!

}

Similarly, for the createPerson-mutation, we’ll have to add a root field to the Mutation type:

type Mutation {

createPerson(name: String!, age: Int!): Person!

}

Notice that this root field takes two arguments as well, the name and the age of the new Person.

Finally, for the subscriptions, we’d have to add the newPerson root field:

type Subscription {

newPerson: Person!

}

Putting it all together, this is the full schema for all the queries and mutation that you have seen in this chapter:

A computer screen shot of a computer code

Description automatically generated

type Query {

allPersons(last: Int): [Person!]!

allPosts(last: Int): [Post!]!

}

type Mutation {

createPerson(name: String!, age: Int!): Person!

updatePerson(id: ID!, name: String!, age: String!): Person!

deletePerson(id: ID!): Person!

}

type Subscription {

newPerson: Person!

}

type Person {

id: ID!

name: String!

age: Int!

posts: [Post!]!

}

type Post {

title: String!

author: Person!

}

# GraphQL – The Big Picture – Architecture

<https://www.howtographql.com/basics/3-big-picture/>

GraphQL has been released only as a specification (<https://facebook.github.io/graphql>). This means that GraphQL is in fact not more than a [long document](https://spec.graphql.org/) that describes in detail the behaviour of a GraphQL server.

## Use Cases

In this section, we’ll walk you through 3 different kinds of architectures that include a GraphQL server:

1. GraphQL server *with a connected database.*
2. GraphQL server to *integrate* existing system: GraphQL server that is a thin layer in front of a number of third party or legacy systems and integrates them through a single GraphQL API.
3. A hybrid approach of a connected database and third party or legacy systems that can all be accessed through the same GraphQL API.

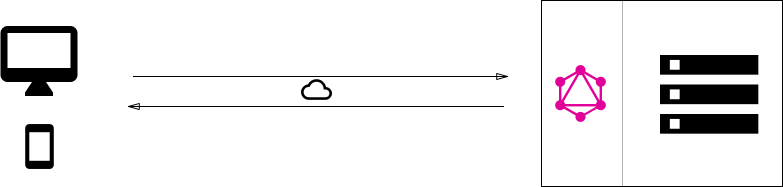
All three architectures represent major use cases of GraphQL and demonstrate the flexibility in terms of the context where it can be used.

### 1. GraphQL server with a connected database

This architecture will be the most common for greenfield projects. In the setup, you have a single (web) server that implements the GraphQL specification. When a query arrives at the GraphQL server, the server reads the query’s payload and fetches the required information from the database. This is called resolving the query. It then constructs the response object [as described in the official specification](https://spec.graphql.org/June2018/#sec-Response) and returns it to the client.

It’s important to note that GraphQL is actually transport-layer agnostic. This means it can potentially be used with any available network protocol. So, it is potentially possible to implement a GraphQL server based on TCP, WebSockets, etc.

GraphQL also doesn’t care about the database or the format that is used to store the data. You could use a SQL database like [AWS Aurora](https://aws.amazon.com/rds/aurora) or a NoSQL database like [MongoDB](https://www.mongodb.com/).



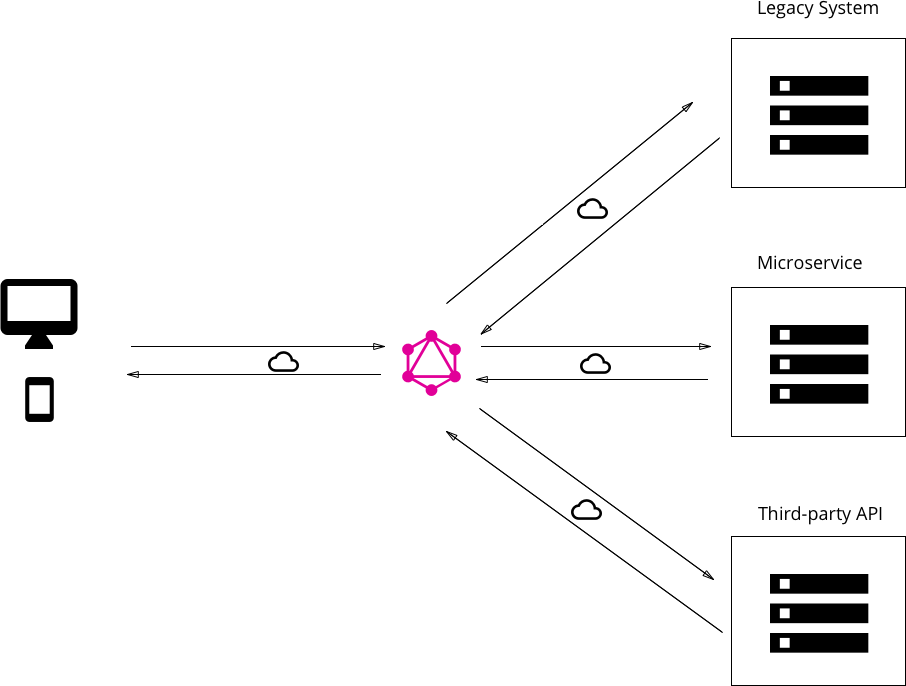
A standard greenfield architecture with one GraphQL server that connects to a single database.

### 2. GraphQL layer that integrates existing systems

Another major use case for GraphQL is the integration of multiple existing systems behind a single, coherent GraphQL API. This is particularly compelling for companies with legacy infrastructures and many different APIs that have grown over years and now impose a high maintenance burden. One major problem with these legacy systems is that they make it practically impossible to build innovative products that need access to multiple systems.

In that context, GraphQL can be used to unify these existing systems and hide their complexity behind a nice GraphQL API. This way, new client applications can be developed that simply talk to the GraphQL server to fetch the data they need. The GraphQL server is then responsible for fetching the data from the existing systems and package it up in the GraphQL response format.

Just like in the previous architecture where the GraphQL server didn’t care about the type of database being used, this time it doesn’t care about the data sources that it needs to fetch the data that’s needed to resolve a query.

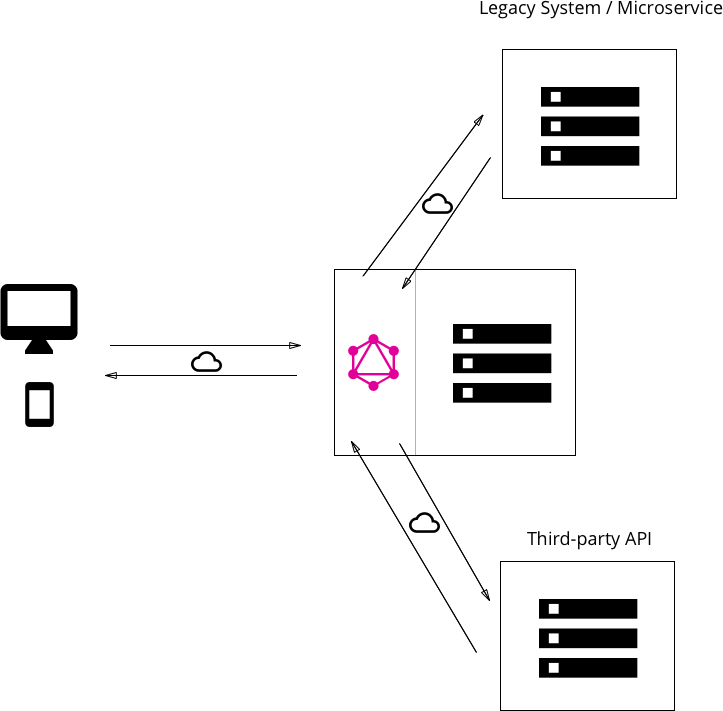


GraphQL allows you to hide the complexity of existing systems, such as microservices, legacy infrastructures or third-party APIs behind a single GraphQL interface.

### 3. Hybrid approach with connected database and integration of existing system

Finally, it’s possible to combine the two approaches and build a GraphQL server that has a connected database but still talks to legacy or third—party systems.

When a query is received by the server, it will resolve it and either retrieve the required data from the connected database or some of the integrated APIs.



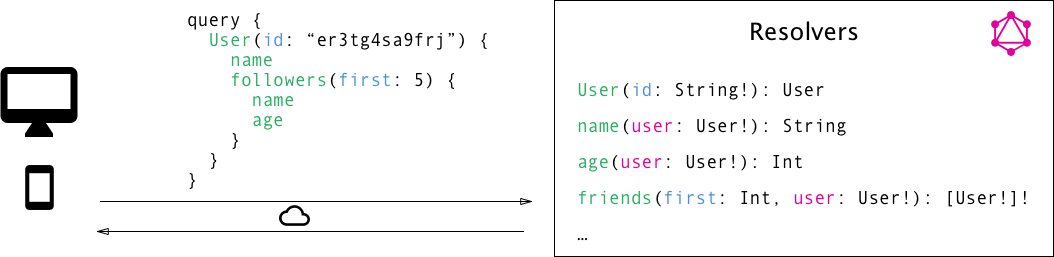
Both approaches can also be combined and the GraphQL server can fetch data from a single database as well as from an existing system - allowing for complete flexibility and pushing all data management complexity to the server.

## Resolver Functions

But how do we gain this flexibility with GraphQL? How is it that it’s such a great fit for these very different kinds of use cases?

As you learned in the previous chapter, the payload of a GraphQL query (or mutation) consists of a set of fields. In the GraphQL server implementation, each of these fields actually corresponds to exactly one function that’s called a [resolver](http://graphql.org/learn/execution/#root-fields-resolvers). The sole purpose of a resolver function is to fetch the data for its field.

When the server receives a query, it will call all the functions for the fields that are specified in the query’s payload. It thus resolves the query and is able to retrieve the correct data for each field. Once all resolvers returned, the server will package data up in the format that was described by the query and send it back to the client.



The above screenshot contains some of the resolved field names. Each field in the query corresponds to a [resolver function](http://graphql.org/learn/execution/#root-fields-resolvers). The GraphQL calls all required resolvers when a query comes in to fetch the specified data.

## GraphQL Client Libraries

GraphQL is particularly great for frontend developers since it completely eliminates many of the inconveniences and shortcomings that are experienced with REST APIs, such as over- and underfetching. Complexity is pushed to the server-side where powerful machines can take care of the heavy computation work. The client doesn’t have to know where the data that it fetches is actually coming from and can use a single, coherent and flexible API.

Let’s consider the major change that’s introduced with GraphQL in going from a rather imperative data fetching approach to a purely declarative one. When fetching data from a REST API, most applications will have to go through the following steps:

1. construct and send HTTP request (e.g., with fetch in JavaScript).
2. receive and parse server response.
3. store data locally (either simply in memory or persistent).
4. display data in the UI.

With the ideal declarative data fetching approach, a client shouldn’t be doing more than the following two steps:

1. describe data requirements.
2. display data in UI.

All the lower-level networking tasks as well as storing the data should be abstracted away and the declaration of data dependencies should be the dominant part.

This is precisely what GraphQL client libraries like Relay or Apollo will enable you to do. They provide the abstraction that you need to be able to focus on the important parts of your application rather than having to deal with the repetitive implementation of infrastructure.

# GraphQL – Advanced Topic – Clients

<https://www.howtographql.com/advanced/0-clients/>

Working with a GraphQL API on the frontend is a great opportunity to develop new abstractions and help implement common functionality on the client-side. Let’s consider some “infrastructure” features that you probably want to have in your app:

* send queries and mutations directly without constructing HTTP requests
* view-layer integration
* caching
* validation and optimization of queries based on the schema

Of course, nothing stops you from using plain HTTP to fetch your data and then shifting all the bits yourself until the right information ends up in your UI. But, GraphQL provides the ability to abstract away a lot of the manual work you’d usually have to do during that process and lets you focus on the real important parts of your app! In the following, we’ll discuss in a bit more detail what these tasks are.

There are two major GraphQL clients available at the moment. The first one is [Apollo Client](https://github.com/apollographql/apollo-client), which is a community-driven effort to build a powerful and flexible GraphQL client for all major development platforms. The second one is called [Relay](https://facebook.github.io/relay/) and it is Facebook’s homegrown GraphQL client that heavily optimizes for performance and is only available on the web.

## Send Queries and Mutations Directly

One major benefit of GraphQL is that it allows you to fetch and update data in a declarative manner. Put differently, we climb up one step higher on the API abstraction ladder and don’t have to deal with low-level networking tasks ourselves anymore.

Where you previously used plain HTTP (like fetch in Javascript or NSURLSession on iOS) to load data from an API, all you need with GraphQL is a query where you declare your data requirements and let the system take care of sending the request and handling the response for you. This is precisely what a GraphQL client will do.

## View Layer Integrations & UI updates

Once the server response is received and handled by the GraphQL client, the requested data somehow needs to end up in your UI. Depending on the platforms and frameworks you’re developing with, there will be different approaches to how UI updates are handled in general.

Taking React as an example, GraphQL clients use the concept of [higher-order components](https://facebook.github.io/react/docs/higher-order-components.html) to fetch the needed data under the hood and make it available in the props of your components. In general, the declarative nature of GraphQL ties in particularly well with [functional reactive programming](https://en.wikipedia.org/wiki/Functional_reactive_programming) techniques. These two form a powerful combination where a view simply declares its data dependencies and the UI is wired up with an FRP layer of your choice.

## Caching Query Results: Concepts and Strategies

In the majority of applications, you’ll want to maintain a cache of the data that was previously fetched from the server. Caching information locally is essential to provide a fluent user experience and also takes the load off your users’ data plans.

Generally, when caching data, the intuition is to put information that’s fetched remotely into a local store from where it can be retrieved later on. With GraphQL, the naive approach would be to simply put the results of GraphQL queries into the store and simply return them whenever the same query is sent. It turns out this approach is very inefficient for most applications.

A more beneficial approach is to normalize the data beforehand. That means that the (potentially nested) query result gets flattened and the store will only contain individual records that can be referenced with a globally unique ID. If you want to learn more about this, the [Apollo blog](https://dev-blog.apollodata.com/the-concepts-of-graphql-bc68bd819be3) has a great write-up on the topic.

## Build-time Schema Validation & Optimizations

Since the schema contains all information about what a client can potentially do with a GraphQL API, there is a great opportunity to validate and potentially optimize the queries that a client wants to send already at build-time.

When the build environment has access to the schema, it can essentially parse all the GraphQL code that’s located in the project and compare it against the information from the schema. This catches typos and other errors before an application gets into the hands of actual users where the consequences of an error would be a lot more drastic.

## Colocating Views and Data Dependencies

A powerful concept of GraphQL is that it allows you to have UI code and data requirements side-by-side. The tight coupling of views and their data dependencies greatly improves the developer experience. The mental overhead of thinking about how the right data ends up in the right parts of the UI is eliminated.

How well colocation works depends on the platform you’re developing on. For example in Javascript applications, it’s possible to actually put data dependencies and UI code into the same file. In Xcode, the Assistant Editor can be used to work on view controllers and graphql code at the same time.

# GraphQL – Advanced Topic – Server

<https://www.howtographql.com/advanced/1-server/>

GraphQL is often explained as a frontend-focused API technology because it enables clients to get data in a much nicer way than before. But the API itself is, of course, implemented on the server side. There are a lot of benefits to be had on the server as well because GraphQL enables the server developer to focus on describing the data available rather than implementing and optimizing specific endpoints.

## GraphQL execution

GraphQL doesn’t just specify a way to describe schemas and a query language to retrieve data from those schemas, but an actual execution algorithm for how those queries are transformed into results. This algorithm is quite simple at its core: The query is traversed field by field, executing “resolvers” for each field. So, let’s say we have the following schema:

type Query {

author(id: ID!): Author

}

type Author {

posts: [Post]

}

type Post {

title: String

content: String

}

The following is a query we would be able to send to a server with that schema:

query {

author(id: "abc") {

posts {

title

content

}

}

}

The first thing to see is that every field in the query can be associated with a type:

query: Query {

author(id: "abc"): Author {

posts: [Post] {

title: String

content: String

}

}

}

Now, we can easily find the resolvers in our server to run for every field. The execution starts at the query type and goes breadth-first. This means we run the resolver for Query.author first. Then, we take the result of that resolver, and pass it into its child, the resolver for Author.posts. At the next level, the result is a list, so in that case, the execution algorithm runs on one item at a time. So the execution works like this:

Query.author(root, { id: 'abc' }, context) -> author

Author.posts(author, null, context) -> posts

for each post in posts

Post.title(post, null, context) -> title

Post.content(post, null, context) -> content

At the end, the execution algorithm puts everything together into the correct shape for the result and returns that.

One thing to note is that most GraphQL server implementations will provide “default resolvers” - so you don’t have to specify a resolver function for every single field. In GraphQL.js, for example, you don’t need to specify resolvers when the parent object of the resolver contains a field with the correct name.

Read more in-depth about GraphQL execution in the [“GraphQL Explained” post](https://dev-blog.apollodata.com/graphql-explained-5844742f195e) on the Apollo blog.

## Batched Resolving

One thing you might notice about the execution strategy above is that it’s somewhat naive. For example, if you have a resolver that fetches from a backend API or database, that backend might get called many times during the execution of one query. Let’s imagine we wanted to get the authors of several posts, like so:

query {

posts {

title

author {

name

avatar

}

}

}

If these are posts on a blog, it’s likely that many of the posts will have the same authors. So if we need to make an API call to get each author object, we might accidentally make multiple requests for the same one. For example:

**fetch**('/authors/1')

**fetch**('/authors/2')

**fetch**('/authors/1')

**fetch**('/authors/2')

**fetch**('/authors/1')

**fetch**('/authors/2')

How do we solve this? By making our fetching a bit smarter. We can wrap our fetching function in a utility that will wait for all of the resolvers to run, then make sure to only fetch each item once:

authorLoader = new AuthorLoader()

*// Queue up a bunch of fetches*

authorLoader.**load**(1);

authorLoader.**load**(2);

authorLoader.**load**(1);

authorLoader.**load**(2);

*// Then, the loader only does the minimal amount of work*

**fetch**('/authors/1');

**fetch**('/authors/2');

Can we do even better? Yes, if our API supports batched requests, we can do only one fetch to the backend, like so:

**fetch**('/authors?ids=1,2')

This can also be encapsulated in the loader above.

In JavaScript, the above strategies can be implemented using a utility called [DataLoader](https://github.com/facebook/dataloader), and there are similar utilities for other languages.

# GraphQL – More Concepts

<https://www.howtographql.com/advanced/2-more-graphql-concepts/>

## Enhancing Reusability with Fragments

Fragments are a handy feature to help to improve the structure and reusability of your GraphQL code. A fragment is a collection of fields on a specific type.

Let’s assume we have the following type:

type User {

name: String!

age: Int!

email: String!

street: String!

zipcode: String!

city: String!

}

Here, we could represent all the information that relates to the user’s physical address into a fragment:

fragment addressDetails on User {

name

street

zipcode

city

}

Now, when writing a query to access the address information of a user, we can use the following syntax to refer to the fragment and save the work to actually spell out the four fields:

{

allUsers {

... addressDetails

}

}

This query is equivalent to writing:

{

allUsers {

name

street

zipcode

city

}

}

## Parameterizing Fields with Arguments

In GraphQL type definitions, each field can take zero or more arguments. Similar to arguments that are passed into functions in typed programming languages, each argument needs to have a name and a type. In GraphQL, it’s also possible to specify default values for arguments.

As an example, let’s consider a part of the schema that we saw in the beginning:

type Query {

allUsers: [User!]!

}

type User {

name: String!

age: Int!

}

We could now add an argument to the allUsers field that allows us to pass an argument to filter users and include only those above a certain age. We also specify a default value so that by default all users will be returned:

type Query {

allUsers(olderThan: Int = -1): [User!]!

}

This olderThan argument can now be passed into the query using the following syntax:

{

allUsers(olderThan: 30) {

name

age

}

}

## Named Query Results with Aliases

One of GraphQL’s major strengths is that it lets you send multiple queries in a single request. However, since the response data is shaped after the structure of the fields being requested, you might run into naming issues when you’re sending multiple queries asking for the same fields:

{

User(id: "1") {

name

}

User(id: "2") {

name

}

}

In fact, this will produce an error with a GraphQL server, since it’s the same field but different arguments. The only way to send a query like that would be to use aliases, i.e. specifying names for the query results:

{

first: User(id: "1") {

name

}

second: User(id: "2") {

name

}

}

In the result, the server would now name each User object according to the specified alias:

{

"first": {

"name": "Alice"

},

"second": {

"name": "Sarah"

}

}

## Advanced SDL

The SDL offers a couple of language features that weren’t discussed in the previous chapter. In the following, we’ll discuss those by practical examples.

### Object & Scalar Types

In GraphQL, there are two different kinds of types.

* Scalar types represent concrete units of data. The GraphQL spec has five predefined scalars: as String, Int, Float, Boolean, and ID.
* Object types have fields that express the properties of that type and are composable. Examples of object types are the User or Post types we saw in the previous section.

In every GraphQL schema, you can define your own scalar and object types. An often cited example for a custom scalar would be a Date type where the implementation needs to define how that type is validated, serialized, and deserialized.

### Enums

GraphQL allows you to define enumerations types (short enums), a language feature to express the semantics of a type that has a fixed set of values. We could thus define a type called Weekday to represent all the days of a week:

enum Weekday {

MONDAY

TUESDAY

WEDNESDAY

THURSDAY

FRIDAY

SATURDAY

SUNDAY

}

**Note** that technically enums are special kinds of scalar types.

### Interface

An interface can be used to describe a type in an abstract way. It allows you to specify a set of fields that any concrete type, which implements this interface, needs to have. In many GraphQL schemas, every type is required to have an id field. Using interfaces, this requirement can be expressed by defining an interface with this field and then making sure that all custom types implement it:

interface Node {

id: ID!

}

type User implements Node {

id: ID!

name: String!

age: Int!

}

### Union Types

Union types can be used to express that a type should be either of a collection of other types. They are best understood by means of an example. Let’s consider the following types:

type Adult {

name: String!

work: String!

}

type Child {

name: String!

school: String!

}

Now, we could define a Person type to be the union of Adult and Child:

union Person = Adult | Child

This brings up a different problem: In a GraphQL query where we ask to retrieve information about a Child but only have a Person type to work with, how do we know whether we can actually access this field?

The answer to this is called conditional fragments:

{

allPersons {

name *# works for `Adult` and `Child`*

... on Child {

school

}

... on Adult {

work

}

}

}

# Tooling and Ecosystem

<https://www.howtographql.com/advanced/3-tooling-and-ecosystem/>

As you probably realized already, the GraphQL ecosystem is growing at an amazing speed right now. One of the reasons that this is happening is because GraphQL makes it really easy for us to develop great tools. In this section, we will see why this is the case, and a few amazing tools we already have in the ecosystem.

If you are familiar with GraphQL basics, you probably know how GraphQL’s Type System allows us to quickly define the surface area of our APIs. It allows developers to clearly define the capabilities of an API, but also to validate incoming queries against a schema.

An amazing thing with GraphQL is that these capabilities are not only known to the server. GraphQL allows clients to ask a server for information about its schema. GraphQL calls this **introspection**.

## Introspection

The designers of the schema already know what the schema looks like but how can clients discover what is accessible through a GraphQL API? We can ask GraphQL for this information by querying the \_\_schema meta-field, which is always available on the root type of a Query per the spec.

query {

\_\_schema {

types {

name

}

}

}

Take this schema definition for example:

type Query {

author(id: ID!): Author

}

type Author {

posts: [Post!]!

}

type Post {

title: String!

}

If we were to send the introspection query mentioned above, we would get the following result:

{

"data": {

"\_\_schema": {

"types": [

{

"name": "Query"

},

{

"name": "Author"

},

{

"name": "Post"

},

{

"name": "ID"

},

{

"name": "String"

},

{

"name": "\_\_Schema"

},

{

"name": "\_\_Type"

},

{

"name": "\_\_TypeKind"

},

{

"name": "\_\_Field"

},

{

"name": "\_\_InputValue"

},

{

"name": "\_\_EnumValue"

},

{

"name": "\_\_Directive"

},

{

"name": "\_\_DirectiveLocation"

}

]

}

}

}

As you can see, we queried for all types on the schema. We get both the object types we defined and scalar types. We can even introspect the introspection types!

There’s much more than name available on introspection types. Here’s another example:

{

\_\_type(name: "Author") {

name

description

}

}

In this example, we query a single type using the \_\_type meta-field and we ask for its name and description. Here’s the result for this query:

{

"data": {

"\_\_type": {

"name": "Author",

"description": "The author of a post.",

}

}

}

As you can see, introspection is an extremely powerful feature of GraphQL, and we’ve only scratched the surface. The specification goes into much more detail about what fields and types are available in the introspection schema.

A lot of tools available in the GraphQL ecosystem use the introspection system to provide amazing features. Think of documentation browsers, autocomplete, code generation, everything is possible! One of the most useful tools you will need as you build and use GraphQL APIs uses introspection heavily. It is called **GraphiQL**.

## GraphQL Playground

[GraphQL Playground](https://github.com/graphcool/graphql-playground) is a powerful “GraphQL IDE” for interactively working with a GraphQL API. It features an editor for GraphQL queries, mutations and subscriptions, equipped with autocompletion and validation as well as a documentation explorer to quickly visualize the structure of a schema (powered by introspection). It also can display your query history or lets you work with multiple GraphQL APIs side-by-side. It also seamlessly integrates with [graphql-config](https://github.com/graphcool/graphql-config).

It is an incredibly powerful tool for development. It allows you to debug and try queries on a GraphQL server without having to write plain GraphQL queries over curl, for example.

# GraphQL – Security

<https://www.howtographql.com/advanced/4-security/>

GraphQL gives enormous power to clients. But with great power come great responsibilities 🕷.

Since clients have the possibility to craft very complex queries, our servers must be ready to handle them properly. These queries may be abusive queries from evil clients, or may simply be very large queries used by legitimate clients. In both of these cases, the client can potentially take your GraphQL server down.

There are a few strategies to mitigate these risks. We will cover them in this chapter in order from the simplest to the most complex, and look at their pros and cons.

## Timeout

The first strategy and the simplest one is using a timeout to defend against large queries. This strategy is the simplest since it does not require the server to know anything about the incoming queries. All the server knows is the maximum time allowed for a query.

For example, a server configured with a 5 seconds timeout would stop the execution of any query that is taking more than 5 seconds to execute.

### Timeout Pros

* Simple to implement.
* Most strategies will still use a timeout as a final protection.

### Timeout Cons

* Damage can already be done even when the timeout kicks in.
* Sometimes hard to implement. Cutting connections after a certain time may result in strange behaviours.

## Maximum Query Depth

As we covered earlier, clients using GraphQL may craft any complex query they want. Since GraphQL schemas are often cyclic graphs, this means a client could craft a query like this one:

query IAmEvil {

author(id: "abc") {

posts {

author {

posts {

author {

posts {

author {

*# that could go on as deep as the client wants!*

}

}

}

}

}

}

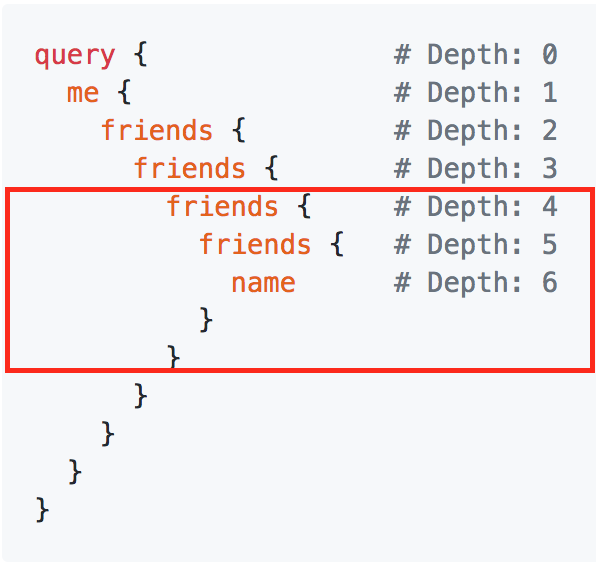
}

}

What if we could prevent clients from abusing query depth like this? Knowing your schema might give you an idea of how deep a legitimate query can go. This is actually possible to implement and is often called Maximum Query Depth.

By analyzing the query document’s abstract syntax tree (AST), a GraphQL server is able to reject or accept a request based on its depth.

Take for example a server configured with a Maximum Query Depth of 3, and the following query document. Everything within the red marker is considered too deep and the query is invalid.



Using graphql-ruby with the max query depth setting, we get the following result:

{

"errors": [

{

"message": "Query has depth of 6, which exceeds max depth of 3"

}

]

}

### Maximum Query Depth Pros

* Since the AST of the document is analyzed statically, the query does not even execute, which adds no load on your GraphQL server.

### Maximum Query Depth Cons

* Depth alone is often not enough to cover all abusive queries. For example, a query requesting an enormous amount of nodes on the root will be very expensive but unlikely to be blocked by a query depth analyzer.

## Query Complexity

Sometimes, the depth of a query is not enough to truly know how large or expensive a GraphQL query will be. In a lot of cases, certain fields in our schema are known to be more complex to compute than others.

Query complexity allows you to define how complex these fields are, and to restrict queries with a maximum complexity. The idea is to define how complex each field is by using a simple number. A common default is to give each field a complexity of 1. Take this query for example:

query {

author(id: "abc") { *# complexity: 1*

posts { *# complexity: 1*

title *# complexity: 1*

}

}

}

A simple addition gives us a total of 3 for the complexity of this query. If we were to set a max complexity of 2 on our schema, this query would fail.

What if the posts field is actually much more complex than the author field? We can set a different complexity to the field. We can even set a different complexity depending on arguments! Let’s take a look at a similar query, where posts has a variable complexity depending on its arguments:

query {

author(id: "abc") { *# complexity: 1*

posts(first: 5) { *# complexity: 5*

title *# complexity: 1*

}

}

}

### Query Complexity Pros

* Covers more cases than a simple query depth.
* Reject queries before executing them by statically analyzing the complexity.

### Query Complexity Cons

* Hard to implement perfectly.
* If complexity is estimated by developers, how do we keep it up to date? How do we find the costs in the first place?
* Mutations are hard to estimate. What if they have a side effect that is hard to measure, like queuing a background job?

## Throttling

The solutions we’ve seen so far are great to stop abusive queries from taking your servers down. The problem with using them alone like this is that they will stop large queries, but won’t stop clients that are making a lot of medium sized queries!

In most APIs, a simple throttle is used to stop clients from requesting resources too often. GraphQL is a bit special because throttling on the number of requests does not really help us. Even a few queries might be too much if they are very large.

In fact, we have no idea what amount of requests is acceptable since they are defined by the clients. So what can we use to throttle clients?

### Throttling Based on Server Time

A good estimate of how expensive a query is the server time it needs to complete. We can use this heuristic to throttle queries. With a good knowledge of your system, you can come up with a maximum server time a client can use over a certain time frame.

We also decide on how much server time is added to a client over time. This is a classic [leaky bucket algorithm](https://en.wikipedia.org/wiki/Leaky_bucket). Note that there are other throttling algorithms out there, but they are out of scope for this chapter. We will use a leaky bucket throttle in the next examples.

Let’s imagine our maximum server time (Bucket Size) allowed is set to 1000ms, that clients gain 100ms of server time per second (Leak Rate) and this mutation:

mutation {

createPost(input: { title: "GraphQL Security" }) {

post {

title

}

}

}

takes on average 200ms to complete. In reality, the time may vary but we’ll assume it always takes 200ms to complete for the sake of this example.

It means that a client calling this operation more than 5 times within 1 second would be blocked until more available server time is added to the client.

After two seconds (100ms is added by second), our client could call the createPost a single time.

As you can see, throttling based on time is a great way to throttle GraphQL queries since complex queries will end up consuming more time meaning you can call them less often, and smaller queries may be called more often since they will be very fast to compute.

It can be good to express these throttling constraints to clients if your GraphQL API is public. In that case, server time is not always the easiest thing to express to clients, and clients cannot really estimate what time their queries will take without trying them first.

Remember the Max Complexity we talked about earlier? What if we throttled based on that instead?

### Throttling Based on Query Complexity

Throttling based on Query Complexity is a great way to work with clients and help them respect the limits of your schema.

Let’s use the same complexity example we used in the Query Complexity section:

query {

author(id: "abc") { *# complexity: 1*

posts { *# complexity: 1*

title *# complexity: 1*

}

}

}

We know that this query has a cost 3 based on complexity. Just like a time throttle, we can come up with a maximum cost (Bucket Size) per time a client can use.

With a maximum cost of 9, our clients could run this query only three times, before the leak rate forbids them to query more.

The principles are the same as our time throttle, but now communicating these limits to clients is much nicer. Clients can even calculate the costs of their queries themselves without needing to estimate server time!

The GitHub public API actually uses this approach to throttle their clients. Take a look at how they express these limits to users: <https://developer.github.com/v4/guides/resource-limitations/>.

## Summary

GraphQL is great to use for clients because it gives them so much more power. But that power also gives them the possibility to abuse your GraphQL server with very expensive queries.

There are many approaches to secure your GraphQL server against these queries, but none of them are bullet proof. It’s important to know what options are available and know their limits so we take the best decisions!

# GraphQL – Common Questions

<https://www.howtographql.com/advanced/5-common-questions/>

## Is GraphQL a Database Technology?

No. GraphQL is often confused with being a database technology. This is a misconception, GraphQL is a query language for APIs - not databases. In that sense it’s database agnostic and can be used with any kind of database or even no database at all.

## Is GraphQL only for React / Javascript Developers?

No. GraphQL is an API technology so it can be used in any context where an API is required.

On the backend, a GraphQL server can be implemented in any programming language that can be used to build a web server. Next to Javascript, there are popular reference implementations for Ruby, Python, Scala, Java, Clojure, Go and .NET.

Since a GraphQL API is usually operated over HTTP, any client that can speak HTTP is able to query data from a GraphQL server.

Note: GraphQL is actually transport layer agnostic, so you could choose protocols other than HTTP to implement your server.

## How to do Server-side Caching?

One common concern with GraphQL, especially when comparing it to REST, are the difficulties to maintain server-side cache. With REST, it’s easy to cache the data for each endpoint, since it’s sure that the structure of the data will not change.

With GraphQL on the other hand, it’s not clear what a client will request next, so putting a caching layer right behind the API doesn’t make a lot of sense.

Server-side caching still is a challenge with GraphQL. More info about caching can be found on the [GraphQL website](http://graphql.org/learn/caching/).

## How to do Authentication and Authorization?

Authentication and authorization are often confused. Authentication describes the process of claiming an identity. That’s what you do when you log in to a service with a username and password, you authenticate yourself. Authorization on the other hand describes permission rules that specify the access rights of individual users and user groups to certain parts of the system.

Authentication in GraphQL can be implemented with common patterns such as [OAuth](https://oauth.net/).

To implement authorization, it is [recommended](http://graphql.org/learn/authorization/) to delegate any data access logic to the business logic layer and not handle it directly in the GraphQL implementation. If you want to have some inspiration on how to implement authorization, you can take a look at this blogpost on [how to implement authorization using GraphQL directives](https://www.prisma.io/blog/graphql-directive-permissions-authorization-made-easy-54c076b5368e).

## How to do Error Handling?

A successful GraphQL query is supposed to return a JSON object with a root field called "data". If the request fails or partially fails (e.g. because the user requesting the data doesn’t have the right access permissions), a second root field called "errors" is added to the response:

{

"data": { ... },

"errors": [ ... ]

}

For more details, you can refer to the [GraphQL specification](http://spec.graphql.org/).

## Does GraphQL Support Offline Usage?

GraphQL is a query language for (web) APIs, and in that sense by definition only works online. However, offline support on the client-side is a valid concern. The caching abilities of Relay and Apollo might already be enough for some use cases, but there isn’t a popular solution for actually persisting stored data yet. You can gain some more insights in the GitHub issues of [Relay](https://github.com/facebook/relay/issues/676) and [Apollo](https://github.com/apollographql/apollo-client/issues/424) where offline support is discussed.

One interesting approach for offline usage and persistence can be found [here](http://www.east5th.co/blog/2017/07/24/offline-graphql-queries-with-redux-offline-and-apollo/).

# React + Apollo Tutorial – Introduction

<https://www.youtube.com/watch?v=5199E50O7SI>

<https://www.howtographql.com/react-apollo/0-introduction/>

**Note:** The final project for this tutorial can be found on [GitHub](https://github.com/howtographql/react-apollo). You can always use it as a reference whenever you get lost throughout the course of the following chapters. Also note that each code block is annotated with a filename. These annotations directly link to the corresponding file on GitHub, so you can clearly see where to put the code and what the end result will look like.

## Overview

In the previous tutorials, we covered the major concepts and benefits of GraphQL. Now is the time to get our hands dirty and start out with an actual project!

We’re going to build a simple clone of [Hackernews](https://news.ycombinator.com/). Here’s a list of the features the app will have:

* Display a list of links
* Search the list of links
* Handle user authentication
* Allow authenticated users to create new links
* Allow authenticated users to upvote links (one vote per link and user)
* Realtime updates when other users upvote a link or create a new one

In this track, we’ll use the following technologies for building the app:

* Frontend:
  + [React](https://facebook.github.io/react/): Library for building user interfaces
  + [Apollo Client 3.2](https://github.com/apollographql/apollo-client): Production-ready, caching GraphQL client
* Backend:
  + [Apollo Server 2.18](https://github.com/apollographql/apollo-server/tree/main/packages/apollo-server): Fully-featured GraphQL Server with focus on easy setup, performance and great developer experience
  + [Prisma](https://www.prisma.io/): Open-source database toolkit that makes it simple to work with relational databases

We’ll create the React project with [create-react-app](https://github.com/facebook/create-react-app), a popular command-line tool that gives us a blank project with all required build configuration already setup.

## Why a GraphQL Client?

In the [Clients](https://www.howtographql.com/advanced/0-clients/) section in the GraphQL part, we already covered the responsibilities of a GraphQL client on a higher level. It’s now time to get more concrete.

In short, we should use a GraphQL client for tasks that are repetitive and agnostic to the app we’re building. For example, being able to send queries and mutations without having to worry about lower level networking details or maintaining a local cache. This is functionality we’ll want in any frontend application that’s talking to a GraphQL server. Why build these features yourself when we can use one of the amazing GraphQL clients out there?

There are several GraphQL client libraries available that all give us varying degrees of control over ongoing GraphQL operations and come with different benefits and drawbacks. For very simple use cases (such as writing scripts), [graphql-request](https://github.com/prisma-labs/graphql-request) might already be enough for our needs. Libraries like it are thin layers around sending HTTP requests to our GraphQL API.

Chances are that you’re writing a somewhat larger application where you want to benefit from caching, optimistic UI updates and other handy features. In these cases you’ll likely want to use a full GraphQL client that handles the lifecycle of all your GraphQL operations. You have the choice between [Apollo Client](https://github.com/apollographql/apollo-client), [Relay](https://facebook.github.io/relay/), and [urql](https://github.com/FormidableLabs/urql).

## Apollo vs Relay vs urql

The most common question heard from people that are getting started with GraphQL on the frontend is which GraphQL client they should use. We’ll try to provide a few hints that’ll help you decide which of these clients is the right one for your next project!

### Relay

[Relay](https://facebook.github.io/relay/) is Facebook’s homegrown GraphQL client that they open-sourced alongside GraphQL in 2015. It incorporates all the learnings that Facebook gathered since they started using GraphQL in 2012. Relay is heavily optimized for performance and tries to reduce network traffic where possible. An interesting side-note is that Relay itself actually started out as a routing framework that eventually got combined with data loading responsibilities. It thus evolved into a powerful data management solution that can be used in JavaScript apps to interface with GraphQL APIs.

The performance benefits of Relay come at the cost of a notable learning curve. Relay is a complex framework and understanding all of its intricacies does require some time. The overall situation in that respect has improved with the release of the 1.0 version, called [Relay Modern](http://facebook.github.io/relay/docs/en/introduction-to-relay.html), but if you’re looking for something to just get started with GraphQL, Relay might not be the right choice just yet.

### Apollo

[Apollo Client](https://github.com/apollographql/apollo-client) is a community-driven effort to build an easy-to-understand, flexible and powerful GraphQL client. Apollo has the ambition to build one library for every major development platform that people use to build web and mobile applications. Right now there is a JavaScript client with bindings for popular frameworks like [React](https://github.com/apollographql/react-apollo), [Angular](https://github.com/apollographql/apollo-angular), [Ember](https://github.com/bgentry/ember-apollo-client) or [Vue](https://github.com/Akryum/vue-apollo) as well as early versions of [iOS](https://github.com/apollographql/apollo-ios) and [Android](https://github.com/apollographql/apollo-android) clients. Apollo is production-ready and has features like caching, optimistic UI, subscription support and more.

### urql

[urql](https://github.com/FormidableLabs/urql) is a more dynamic approach on GraphQL clients and a newer project compared to Relay and Apollo. While it’s highly focused on React, at its core it focuses on simplicity and extensibility. It comes with the barebones to build an efficient GraphQL client, but gives you full control over how it processes GraphQL operations and results via “Exchanges”. Together with the first-party exchange [@urql/exchange-graphcache](https://github.com/FormidableLabs/urql-exchange-graphcache) it is basically equivalent in functionality with Apollo, but with a smaller footprint and a very focused API.

# GraphQL Example with Node.JS and Apollo

**Source**: dotNetFullStackDemos\GraphQL.NET\graphql-server-example

We will create an GraphQL server using Node.JS and Apollo Server. Apollo will also give us an Apollo Explorer where we can type and run our GraphQL queries

## Setup Apollo Server

To setup Apollo server, follow the instructions at this url: <https://www.apollographql.com/docs/apollo-server/getting-started>

This tutorial assumes that you are familiar with the command line and JavaScript and have installed a recent Node.js (v14.16.0+) version. Additionally, for those interested, this tutorial includes an optional section describing how to set up Apollo Server with TypeScript.

### [Step 1: Create a new project](https://www.apollographql.com/docs/apollo-server/getting-started#step-1-create-a-new-project)

1. From your preferred development directory, create a directory for a new project and cd into it:

mkdir graphql-server-example

cd graphql-server-example

Initialize a new Node.js project with npm (or another package manager you prefer, such as Yarn):

npm init --yes && npm pkg set type="module"

This getting started guide sets up a project using ES Modules, which simplifies our examples and allows us to use top-level await.

Your project directory now contains a package.json file.

### [Step 2: Install dependencies](https://www.apollographql.com/docs/apollo-server/getting-started#step-2-install-dependencies)

Applications that run Apollo Server require two top-level dependencies:

* [graphql](https://npm.im/graphql) (also known as graphql-js) is the library that implements the core GraphQL parsing and execution algorithms.
* [@apollo/server](https://www.npmjs.com/package/@apollo/server) is the main library for Apollo Server itself. Apollo Server knows how to turn HTTP requests and responses into GraphQL operations and run them in an extensible context with support for plugins and other features.

Run the following command to install both of these packages and save them in your project's node\_modules directory:

npm install @apollo/server graphql

Follow the instructions below to set up with either TypeScript or JavaScript:

#### Set up with TypeScript (\*\*Recommended\*\*)

1. Create a src directory with an empty index.ts file to contain **all** of the code for our example application:

mkdir src

touch src/index.ts

1. Run the following command to install the typescript and @types/node packages into your project's dev dependencies:

npm install --save-dev typescript @types/node

1. Next, create a tsconfig.json file in your project:

touch tsconfig.json

1. The tsconfig.json file enables you to configure how TypeScript will compile your code. Add the following configuration to your tsconfig.json file:

tsconfig.json

{

"compilerOptions": {

"rootDirs": ["src"],

"outDir": "dist",

"lib": ["es2020"],

"target": "es2020",

"module": "esnext",

"moduleResolution": "node",

"esModuleInterop": true,

"types": ["node"]

}

}

1. Finally, replace the default scripts entry in your package.json file with the following type and scripts entries:

package.json

{

// ...etc.

"type": "module",

"scripts": {

"compile": "tsc",

"start": "npm run compile && node ./dist/index.js"

}

// other dependencies

}

The above start script tells TypeScript to compile your code into JavaScript before using node to run that compiled code. Setting your project's [type to module](https://nodejs.org/api/packages.html#approach-1-use-an-es-module-wrapper) loads your JavaScript files as ES modules, enabling you to use top-level [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) calls.

1. You can now run npm start, which should successfully compile and run your empty index.ts file, printing something like this:

$ npm start

> graphql-server-example@1.0.0 start

> npm run compile && node ./dist/index.js

### [Step 3: Define your GraphQL schema](https://www.apollographql.com/docs/apollo-server/getting-started#step-3-define-your-graphql-schema)

Every GraphQL server (including Apollo Server) uses a **schema** to define the structure of data that clients can query. In this example, we'll create a server for querying a collection of books by title and author.

**typeDefs** are definitions of the different types of data we want to expose on our graph. For example, we might make a typeDef for an author data type and specify different fields that author might have, like a name, an avatar url, a bio etc. And we might one for a game, which is a title, a price, a platform etc. So, these are the different data that we want to make available on the graph so that a user can eventually query.

And the combination of all these types and the relationship to other types and the kinds of queries that can be made to combine up to make something called a **schema**. The schema is something that describes the shape of the graph and the data available on it. Normally, your GraphQL schema, the data that is available on the graph will be fairly similar to the data you’re storing in your application database.

Open index.ts in your preferred code editor and paste the following into it:

import { ApolloServer } from '@apollo/server';

import { startStandaloneServer } from '@apollo/server/standalone';

// A schema is a collection of type definitions (hence "typeDefs")

// that together define the "shape" of queries that are executed against

// your data.

const typeDefs = `#graphql

# Comments in GraphQL strings (such as this one) start with the hash (#) symbol.

# This "Book" type defines the queryable fields for every book in our data source.

type Book {

title: String

author: String

}

# The "Query" type is special: it lists all of the available queries that

# clients can execute, along with the return type for each. In this

# case, the "books" query returns an array of zero or more Books (defined above).

type Query {

books: [Book]

}

`;

**Note**: Adding #graphql to the beginning of a [template literal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) provides GraphQL syntax highlighting in supporting IDEs. *Install the GraphQL: Syntax Highlighting extension in VS Code*.

This snippet defines a simple, valid GraphQL schema. Clients will be able to execute a query named books, and our server will return an array of zero or more Books.

GraphQL has five basic scalar types:

* int
* float
* string
* bool
* ID – used as a key for data objects.

### [Step 4: Define your data set](https://www.apollographql.com/docs/apollo-server/getting-started#step-4-define-your-data-set)

Now that we've defined the *structure* of our data, we can define the data itself.

Apollo Server can fetch data from any source you connect to (including a database, a REST API, a static object storage service, or even another GraphQL server). For the purposes of this tutorial, we'll hardcode our example data.

Add the following to the bottom of your index.ts file:

const books = [

{

title: 'The Awakening',

author: 'Kate Chopin',

},

{

title: 'City of Glass',

author: 'Paul Auster',

},

];

This snippet defines a simple data set that clients can query. Notice that the two objects in the array each match the structure of the Book type we defined in our schema.

### [Step 5: Define a resolver](https://www.apollographql.com/docs/apollo-server/getting-started#step-5-define-a-resolver)

We've defined our data set, but Apollo Server doesn't know that it should *use* that data set when it's executing a query. To fix this, we create a **resolver**.

Resolvers tell Apollo Server *how* to fetch the data associated with a particular type. Because our Book array is hardcoded, the corresponding resolver is straightforward.

Add the following to the bottom of your index.ts file:

// Resolvers define how to fetch the types defined in your schema.

// This resolver retrieves books from the "books" array above.

const resolvers = {

Query: {

books: () => books,

},

};

### [Step 6: Create an instance of ApolloServer](https://www.apollographql.com/docs/apollo-server/getting-started#step-6-create-an-instance-of-apolloserver)

We've defined our schema, data set, and resolver. Now we need to provide this information to Apollo Server when we initialize it.

Add the following to the bottom of your index.ts file:

// The ApolloServer constructor requires two parameters: your schema

// definition and your set of resolvers.

const server = new ApolloServer({

typeDefs,

resolvers,

});

// Passing an ApolloServer instance to the `startStandaloneServer` function:

// 1. creates an Express app

// 2. installs your ApolloServer instance as middleware

// 3. prepares your app to handle incoming requests

const { url } = await startStandaloneServer(server, {

listen: { port: 4000 },

});

console.log(`🚀 Server ready at: ${url}`);

Here, the ApolloServer takes in an object as an argument. That object expects two properties:

* typeDefs – short for type definitions. These are descriptions of our data types and the relationship they have with other data types.
* resolvers – a bunch of resolver functions that determine how we respond to queries for different data on the graph.

### [Step 7: Start the server](https://www.apollographql.com/docs/apollo-server/getting-started#step-7-start-the-server)

We're ready to start our server! Run the following from your project's root directory:

npm start

You should now see the following output at the bottom of your terminal:

🚀 Server ready at: http://localhost:4000/

We're up and running!

### [Step 8: Execute your first query](https://www.apollographql.com/docs/apollo-server/getting-started#step-8-execute-your-first-query)

We can now execute GraphQL queries on our server. To execute our first query, we can use [**Apollo Sandbox**](https://www.apollographql.com/docs/graphos/explorer/sandbox/).

Visit http://localhost:4000 in your browser, which will open the Apollo Sandbox:

A screenshot of a computer

Description automatically generated

The Sandbox UI includes:

* An Operations panel for writing and executing queries (in the middle)
* A Response panel for viewing query results (on the right)
* Tabs for schema exploration, search, and settings (on the left)
* A URL bar for connecting to other GraphQL servers (in the upper left)

Our server supports a single query named books. Let's execute it!

Here's a GraphQL **query string** for executing the books query:

query ExampleQuery {

  books {

    title,

    author

  }

}

Paste this string into the Operations panel and click the blue button in the upper right. The results (from our hardcoded data set) appear in the Response panel:



One of the most important concepts of GraphQL is that clients can choose to query *only for the fields they need*. Delete author from the query string and execute it again. The response updates to include only the title field for each book!

## Game, Review and Author

Now let’s start with Game and define its properties.

~~Create a new file named src/schema.js and~~ In the index.ts file, add this code, where the id, title and platform properties are identified as NOT NULL using the exclamation (!) symbol:

export const typeDefs = `*#graphql*

    type Game {

        id: ID!,

        title: String!,

        platform: [String!]!

    }

`

The platform property has an ! outside the square brackets indicating we must have value for platform, which is an array of strings, but the elements inside the array can be nullable if we do not specify the ! inside the square brackets.

Let’s add the Review and Author types:

export const typeDefs = `*#graphql*

    type Game {

        id: ID!,

        title: String!,

        platform: [String]!

    }

    type Review {

        id: ID!

        rating: Int!

        content: String!

    }

    type Author {

        id: ID!

        name: String!

        verfied: Boolean!

    }

`

We also need a special type, let’s call it Query. The Query type is something that every GraphQL schema that you make needs to have. It’s not optional. And its job is to define the entry points to the graph and specify the return types of those entry points.

So, for e.g., if I want users to be able to query the Review data that we have and get back a list of reviews, then I need to specify that inside this Query type. So, we make a field called reviews and then tell GraphQL that we expect to return type of this entry point to be a list of Reviews.

type Query {

    reviews: [Reviews]

}

Now, if we left the Query type like this, we’re essentially saying we only want to expose that one single entry point to the graph, meaning a user would only be able to enter the graph at this point (reviews) and then they’d be free to navigate around the graph to eventually get related data, but they wouldn’t be able to jump in at any other point, whether that be a single review instead of a list of reviews or an author or a game because we have not specified those entry points right here.

So, Query type is our way of gate-keeping entry on to the graph and deciding where the user can jump into it initially. So, let’s make some more entry points:

type Query {

    reviews: [Review]

    games: [Game]

    author: [Author]

}

We will see how to land on single one’s later.

So, the entire typeDefs will be as follows:

export const typeDefs = `*#graphql*

    type Game {

        id: ID!,

        title: String!,

        platform: [String]!

    }

    type Review {

        id: ID!

        rating: Int!

        content: String!

    }

    type Author {

        id: ID!

        name: String!

        verfied: Boolean!

    }

    type Query {

        reviews: [Review]

        games: [Game]

        author: [Author]

    }

`

Next thing we need to do is pass all these type definitions into the Apollo Server that we created earlier.

~~First, rename the typeDefs constant in index.ts to typeDefsBook.~~

~~Now, import the typeDefs from schema.ts into the index.ts and add it to the Apollo server:~~

*~~// types.~~*

~~import { typeDefs } from './schema';~~

const server = new ApolloServer({

    typeDefs

  });

The second parameter is for resolvers, which are basically a bunch of resolver functions. They are there for us to handle any incoming requests and return data to the client. At the moment, all we have done is define what the graph looks like in terms of the data types that we have and the entry points. But we have not yet said how we want to handle requests or queries for that data and that’s what the resolver functions are for.

So, you can think of the schema and the type definitions as like a map for Apollo to structure the graph, but they don’t actually handle any queries. And, then we make resolver functions to handle the queries based on our schema type.

## Configure the Data

We don’t have a DB right now. Instead, we will use some local data stored in a variable or in another file. ~~Create a file called src/\_db.js and~~ In the index.ts file, add some data to it as follows:

let games = [

    {id: '1', title: 'Zelda, Tears of the Kingdom', platform: ['Switch']},

    {id: '2', title: 'Final Fantasy 7 Remake', platform: ['PS5', 'Xbox']},

    {id: '3', title: 'Elden Ring', platform: ['PS5', 'Xbox', 'PC']},

    {id: '4', title: 'Mario Kart', platform: ['Switch']},

    {id: '5', title: 'Pokemon Scarlet', platform: ['PS5', 'Xbox', 'PC']},

  ]

let authors = [

    {id: '1', name: 'mario', verified: true},

    {id: '2', name: 'yoshi', verified: false},

    {id: '3', name: 'peach', verified: true},

]

let reviews = [

    {id: '1', rating: 9, content: 'lorem\_ipsum', author\_id: '1', game\_id: '2' },

    {id: '2', rating: 10, content: 'lorem\_ipsum', author\_id: '2', game\_id: '1' },

    {id: '3', rating: 7, content: 'lorem\_ipsum', author\_id: '3', game\_id: '3' },

    {id: '4', rating: 5, content: 'lorem\_ipsum', author\_id: '2', game\_id: '4' },

    {id: '5', rating: 8, content: 'lorem\_ipsum', author\_id: '1', game\_id: '5' },

    {id: '6', rating: 7, content: 'lorem\_ipsum', author\_id: '3', game\_id: '2' },

    {id: '7', rating: 10, content: 'lorem\_ipsum', author\_id: '3', game\_id: '1' },

]

~~export default { games, authors, reviews }~~

**Note**: Ignore the author\_id and game\_id properties in the reviews array for now.

## Resolver Functions

Now, let’s make resolve functions for these. In the index.ts file, modify the resolvers variable and inside this object, we make resolver functions for each different type that we define. We want to create resolve functions for the query type because that root query type is where we define entry points to the graph and specify what data should be returned for them.

So, let’s create resolve functions for games, authors and reviews. Create an object named Query in the resolver variable – it must be exactly the same that is define in the typeDefs variable – which is an object as well. In this Query object, define resolver functions for each of the properties defined on our root query type.

const resolvers = {

    Query: {

        games() {

            return games

        },

        reviews() {

            return reviews

        },

        authors() {

            return authors

        }

    },

};

Now, make sure we pass this resolvers object to the Apollo Server:

const server = new ApolloServer({

    typeDefs,

    resolvers,

});

Run the project with npm start

Navigate to <http://localhost:4000> and run the following queries in separate tabs:

query GamesQuery {

  allGames {

    id,

    title

    platform

  }

}

query AuthorsQuery {

  allAuthors {

    id

    name

    verified

  }

}

query ReviewsQuery {

  allReviews {

    id

    rating

    content

  }

}

## Query Variables

What if a user wants to send a query for a single review or game or author?

We will have to create entry points and resolvers for this. So, let’s create entry points in the Query object of the typeDefs variable:

review(*id*: ID!): Review

Here, we are creating an entry point for a single review object to be retrieved by the query, but for that, we require an ID, which is the Query Variable of type ID and since it is required, we suffix it with the ! mark.

Now, let’s make a resolver function for this query. In the Query property inside the resolvers variable, add a new function as follows:

review(*parent*, *args, context*) {

:

}

Here, the first argument refers to the parent resolver in a resolver chain. This usually is useful and makes sense when we deal with related data and nested queries. Since we don’t need that in this function, we will rename it to an underscore as follows:

review(*\_*, *args, context*) {

:

}

The 2nd argument that we do need is called args. It is here that we can access any variable that we pass to the query.

The third is a context object, which we can use for supplying context values across all of our resolvers such as authentication information etc. We don’t need that for now, so we will remove that.

review(*\_*, *args*) {

return reviews.find((*review*) => review.id === args.id)

},

The function just iterates through the data one row at a time, checks if the id matches and if yes, returns that row.

Run the code and execute the following query to check if it works:

query SingleReviewQuery($id: ID!) {

  review(id: $id) {

    rating

    content

  }

}

In Apollo Explorer, in the Variables tab at the bottom, enter the following JSON:

{

  "id": "2"

}

When you execute the query, Apollo will pass this JSON to the query, which will extract the id value and pass it to the $id parameter of the top level query SingleReviewQuery, which will then be passed on to the actual query review. It will fetch the data for the given id and display the result.

Now add entry points and resolvers for the remaining two types – games and authors – and execute their respective queries similarly. Here is the code:

const typeDefs = `*#graphql*

*:*

    type Query {

        books: [Book]

        reviews: [Review]

        review(*id*: ID!): Review

        games: [Game]

        game(*id*: ID!) : Game

        authors: [Author]

        author(*id*: ID!) : Author

    }

`;

const resolvers = {

    Query: {

        games() {

            return games

        },

        reviews() {

            return reviews

        },

        authors() {

            return authors

        },

        review(*\_*, *args*) {

            return reviews.find((*review*) => review.id === args.id)

        },

        game(*\_*, *args*) {

            return games.find((*game*) => game.id === args.id)

        },

        author(*\_*, *args*) {

            return authors.find((*author*) => author.id === args.id)

        }

    },

};

query SingleGameQuery($id: ID!) {

  game(id: $id) {

    title

    platform

  }

}

query SingleAuthorQuery($id: ID!) {

  author(id: $id) {

    name

    verified

  }

}

## Related Data

Let’s look at the reviews sample data:

let reviews = [

    {id: '1', rating: 9, content: 'lorem\_ipsum', author\_id: '1', game\_id: '2' },

    {id: '2', rating: 10, content: 'lorem\_ipsum', author\_id: '2', game\_id: '1' },

    {id: '3', rating: 7, content: 'lorem\_ipsum', author\_id: '3', game\_id: '3' },

    {id: '4', rating: 5, content: 'lorem\_ipsum', author\_id: '2', game\_id: '4' },

    {id: '5', rating: 8, content: 'lorem\_ipsum', author\_id: '1', game\_id: '5' },

    {id: '6', rating: 7, content: 'lorem\_ipsum', author\_id: '3', game\_id: '2' },

    {id: '7', rating: 10, content: 'lorem\_ipsum', author\_id: '3', game\_id: '1' },

]

We have author\_id and game\_id that map the review to specific author and game. So, given a review, we should be able to find out who wrote the review and for which game. And vice versa as well – given a game\_id (from the games data), we can determine all the reviews for that game.

We need a way to find these relations in the schemas so that Apollo knows to make our graph that way. So, to begin with, let’s set the relationships for our schema as follows (*note the highlighted parts*):

export const typeDefs = `*#graphql*

    type Review {

        id: ID!

        rating: Int!

        content: String!

game: Game!

author: Author!

    }

    type Game {

        id: ID!,

        title: String!,

        platform: [String]!

        reviews: [Review!]

    }

    type Author {

        id: ID!

        name: String!

        verfied: Boolean!

        reviews: [Review!]

    }

    type Query {

        books: [Book]

        reviews: [Review]

        review(*id*: ID!): Review

        games: [Game]

        game(*id*: ID!) : Game

        authors: [Author]

        author(*id*: ID!) : Author

    }

`

Here, in the Review schema, we know that each review is supposed to be for a game, and hence the game: Game! Property. And the review is by an author, hence the author: Author! property. Both properties are required because if there is a review, then it has to be for a Game and it has to be by an Author.

Similarly, in the Game schema, we have reviews: [Review!]. This indicates that there can be multiple reviews for each game (hence an array – [Review!]) and if there is a review for a game, then it has to be of type Review, hence the ! suffixed for the [Review!]. But, it is not mandatory to have a review for a Game, and hence there is no ! after the array definition.

And similarly for the Author schema, we have reviews: [Review!].

Now, we also need to make the resolver functions to resolve any nested queries for the related data. For e.g., I may query for a game and all the related reviews for that game. The query may look something like this:

query GameQuery($id: ID!) {

  game(id: $id) {

    title

    reviews {

      rating

      content

    }

  }

}

Here, the single jumping in point is for a single game, but then we also ask for the reviews related to that game along with the rating and content for that review.

As of now, Apollo does not know how to handle that nested query for the reviews inside the specific game. The only way it knows how to resolve reviews currently is either by grabbing all of them, or just by grabbing one of them based on the ID. And these are both root queries defined in the query type in our schema. So, it doesn’t know how to get a subset of reviews based on the ID of a particular game. We don’t have a resolver for that.

So, the way we make the resolver is not by making it inside the query object because these are resolvers for entry points to the graph as defined by the query type that we made in schema. So instead, because this nested request is associated with a game object, we make a new property inside the resolvers object called Game, which is also an object. And then inside this we can make a resolver function called reviews where we can tell Apollo how to get all the reviews based on the pairing query for the single game.

const resolvers = {

    Query: {

:

:

    },

Game: {

        reviews(*parent*) {

            return reviews.filter((*r*) => r.game\_id === parent.id)

        }

    }

};

The reviews resolver function takes in an argument called parent. As mentioned earlier, we didn’t need this before. However, now do need it. So, the way this is going to work is because the entry point of our query is a single game, Apollo will run that initial resolver function inside the query object to get that single game. Then, to resolve the reviews for that game, it’s going to look to the Game object, since that’s what we grabbed, and then its going to look for the reviews resolver inside that to grab the reviews.

So, its inside this Game function that we tell Apollo how to do that. But how do we know what game we’re getting reviews for? Well, we can access the ID of the game via the parent argument because parent argument is a reference to the value returned by the previous or parent resolver. Now, in our case, that’s going to be the game resolver function. So, the parent argument will basically be a game object and that game object is going to have an ID, which we can then use.

So, we can use the ID to return all the reviews associated with that game ID.

Let’s try running the query for different game ids.

query GameQuery($id: ID!) {

  game(id: $id) {

    title

    reviews {

      rating

      content

    }

  }

}

So, now that we have resolved how to find reviews for a given game, but reviews can also be associated with an author. And if we look at the Author schema, each author could have a list of reviews as well (*just like a Game*). So, let’s do the same thing for Author.

Just like we had a Game property in the resolvers object, this time we will create an Author property and then the reviews resolver function for this where we are taking the parent and inside this function, we do the same thing as we did for the Game reviews based on the author\_id of the review.

const resolvers = {

    Query: {

:

:

    },

Game: {

        reviews(*parent*) {

            return reviews.filter((*r*) => r.game\_id === parent.id)

        }

    },

Author: {

        reviews(*parent*) {

            return reviews.filter((*r*) => r.author\_id === parent.id)

        }

    },

};

Try running the following query and see if it works:

query GameAuthorQuery($id: ID!) {

  game(id: $id) {

    title

    reviews {

      rating

      content

      author {

        name

      }

    }

  }

}

It won’t work just yet because there is one more set of resolvers that we need for the Review. So, imagine if we select a single review, as per our schema, each Review has an associated Game and an Author. So, they would be nested queries. So, we need to make a resolver function to get the game associated with that review and also the author. So, we need two resolver functions for each Review:

1. To fetch the author for the review.
2. To fetch the game for the review.

const resolvers = {

    Query: {

:

:

    },

Game: {

:

    },

Author: {

:

    },

    Review: {

        author(*parent*) {

            return authors.find((*a*) => a.id === parent.author\_id)

        },

        game(*parent*) {

            return games.find((*g*) => g.id === parent.game\_id)

        },

    }

};

Now try running the same query again and it should work!

Try the following queries as well:

query AuthorQuery($id: ID!) {

  author(id: $id) {

    name

    reviews {

      rating

      content

    }

  }

}

query ReviewQuery($id: ID!) {

  review(id: $id) {

    rating,

    content,

    game {

      title,

      platform

    },

    author {

      name,

      verified

    }

  }

}

Works! Perfect!

We can also try this query where we say, get all the Reviews and for each Review, get the Game associated with it and also get all the Reviews for the Game and their associated Author:

query ReviewQuery($id: ID!) {

  review(id: $id) {

    rating

    game {

      title,

      platform,

      reviews {

        rating,

        content,

        author {

          name

        }

      }

    }

  }

}

Awesome!

## Mutations – Adding & Deleting Data

For now, we can only fetch the data. We cannot add, edit or delete any of the data. That’s where mutations come into play. Mutation is a generic term in GraphQL for any kind of change that we would make to the data.

So, first we need to define our allowed mutations in the schema (*in the* typeDefs *variable*) by making a new type, which is called mutation. And it is inside this type that we can then decide how users can mutate any data. For e.g., I might want to expose a mutation called deleteGame, and for that mutation, we need an ID argument to say what Game should be deleted. We also specify the return type as well after a user makes this mutation much like we did for the root queries. So, for e.g., once a user deletes a Game, we might want to send back an updated list of all the games after that one has been removed. So, we use an array of Game objects.

const typeDefs = *`#graphql*

:

:

    Review: {

:

    },

    Mutation: {

        deleteGame(*\_*, *args*) {

            games = games.filter((*game*) => game.id !== args.id)

            return games

        }

    }

};

To run a mutation query from the front-end, we use the work “mutation” instead of “query.” Let’s try it with this mutation to delete a Game:

mutation DeleteGame($id: ID!) {

  deleteGame(id: $id) {

    id,

    title,

    platform

  }

}

Now, let’s try adding a new Game.

First, we need to go to the Mutation type in the schema and add the mutation called addGame that returns a single Game object, the one that we just created. It will also accept some arguments that make up a Game – minus the ID as that will be auto-generated in the resolver for this mutation later.

We could add each of the properties as an argument, but instead we could make a new special input type in our schema, which allows us got group together several arguments into one type, and then that can be used as a single argument elsewhere, like in this mutation. So, the way we do that is by creating an input instead of type, which says to GraphQL that this isn’t an actual type of data, but more of a collection of fields that we can use in a mutation as a single argument.

const typeDefs = *`#graphql*

:

:

    type Mutation {

        deleteGame(*id*: ID!): [Game]

    },

    input AddGameInput {

        title: String!

        platform: [String!]!

    }

`;

We don’t specify the Reviews because we are not making a review here.

Now, we create a mutation called addGame that takes in a variable called game that is of type AddGameInput and it is required and it returns the newly created Game object. Here’s how we do this:

const typeDefs = *`#graphql*

:

:

    type Mutation {

        deleteGame(*id*: ID!): [Game]

        addGame(*game*: AddGameInput!): Game

    },

    input AddGameInput {

        title: String!

        platform: [String!]!

    }

`;

Now that we have this sorted, we have to add this mutation to the Mutation object inside the resolvers object ss follows:

const resolvers = {

:

:

    Review: {

:

    },

    Mutation: {

        deleteGame(*\_*, *args*) {

            games = games.filter((*game*) => game.id !== args.id)

            return games

        },

        addGame(*\_*, *args*) {

            let game = {

                ...args.game,

                id: Math.floor(Math.random() \* 10000).toString()

            }

            games.push(game)

            return game

        }

    }

};

Here, the args property will have the game object of type AddGameInput, which will have the properties required to create a new Game.

What we are doing here is make a new Game object and add it to the games array. So, we make the game object first and then we spread out …args.game because that is the name of the variable that gets passed to args ,and on that will be the two properties required to create the game – the title and the platform. We also need an id property, which will be an auto-generated number. We use the Math.random() function to generate a random number between 0 and 1, multiply it by 10000. This will give us a decimal, so we use the Math.floor() function to make it an integer. This will give us a number between 1 and 10000. And then we convert it to string with toString().

Then, we add the new Game object to the array and return that same object back to the user.

Now, let’s try this from the front-end with the following input values:

{

  "game": {

    "title": "Halo 4",

    "platform": ["XBox", "PS5"]

  }

}

mutation AddGameMutation($game: AddGameInput!) {

  addGame(game: $game) {

    id,

    title,

    platform

  }

}

And then run this query to see if the game was indeed added:

query Games {

  games {

    id,

    title

    platform

  }

}

Works!

## Mutations – Updating Data

Now that we have seen how to create mutations for adding and deleting data, let’s see how to update data with mutations.

Let’s create a new mutation first called updateGame. Here, we need the ID of the game to update and the kind of edits required. For these edits, we will create a new input type, just like we did for AddGameInput and pass this as the second argument to the updateGame mutation. And this mutation returns the updated Game object.

const typeDefs = *`#graphql*

:

:

    type Mutation {

        deleteGame(*id*: ID!): [Game]

        addGame(*game*: AddGameInput!): Game

        updateGame(*id*: ID, *edits*: EditGameInput!): Game

    },

    input EditGameInput {

        title: String

        platform: [String!]

    },

    input AddGameInput {

        title: String!

        platform: [String!]!

    }

`;

Now, if you notice, the two input types are almost the same. So, why didn’t we reuse the AddGameInput? The difference is that for the update type, the two properties are not required because you might want to update only a specific field and not all of them.

Now, we can create a resolver function for this update game mutation.

const resolvers = {

:

:

    Review: {

:

    },

    Mutation: {

        deleteGame(*\_*, *args*) {

            games = games.filter((*game*) => game.id !== args.id)

            return games

        },

        addGame(*\_*, *args*) {

            let game = {

                ...args.game,

                id: Math.floor(Math.random() \* 10000).toString()

            }

            games.push(game)

            return game

        },

        updateGame(*\_*, *args*) {

            games = games.map((*g*) =>{

                if(g.id === args.id) {

                    return {

                        ...g,

                        ...args.edits

                    }

                } else {

                    return g

                }

            })

            let game = games.find((*game*) => game.id === args.id)

            return game

        }

    }

};

Here, we are mapping through the games array and creating a new array out of it. We fire a function for each item in the array, and for each item we check if the ID of that game is equal to the ID (args.id) of the game we want to edit. If it does match, then we return the current object “g” and spread those with the properties in them, and we are also spreading “args.edit”. So, for e.g., if we are going to update the title, then it is going to override the title that we are spreading with “g”. And this object then goes into the array then. If we don’t get a match, then we don’t change anything and return the original “g” object back which goes into the new array.

And then, we find the Game object we edited from the new array and return that back to the user.

So, let’s try it out with the following input values:

{

  "edits": {

    "title": "Halo Infinite"

  },

  "id": "1037"

}

mutation EditGameMutation($id: ID!, $edits: EditGameInput!) {

  updateGame(id: $id, edits: $edits) {

    id,

    title,

    platform

  }

}

And then run this query to see if the game was indeed added:

query Games {

  games {

    id,

    title

    platform

  }

}

Works!

Now, let’s try editing the platforms with the following input values:

{

  "edits": {

    "platform": ["XBox","PS5", "Switch"]

  },

  "id": "1037"

}

Run the same edit mutation. Works!

## Calling the GraphQL API Using curl

curl \

-X POST \

-H "Content-Type: application/json" \

--data '{ "query": "{ games { id, title, platform } }" }' \

http://localhost:4000/graphql

## Calling the GraphQL API Using Postman

A screenshot of a computer

Description automatically generated

## Calling the GraphQL API Using fetch()

* Create a new folder and navigate into this new folder.
* Run npm init --yes && npm pkg set type="module".
* Creates a package.json file.
* Run npm install node-fetch.
* Run touch index.js.
* Copy the following code into index.js.

import fetch from 'node-fetch';

fetch('http://localhost:4000', {

  method: 'POST',

  headers: { 'Content-Type': 'application/json' },

  body: JSON.stringify({ query: `

    query {

        games {

            id,

            title,

            platform

        }

    }`

  }),

})

.then(*res* => res.json())

.then(*res* => console.log(res.data));

* In package.json file, replace the “scripts” section with this:

  "scripts": {

    "start": "node index.js"

  },

* Run the app with npm start.
* Check the results.

# React + Apollo – Getting Started

<https://www.howtographql.com/react-apollo/1-getting-started/>

Since this is a frontend track, we’re not going to spend any time implementing the backend. Instead, we’ll use the server from the [Node tutorial](https://www.howtographql.com/graphql-js/0-introduction).

Once our React application is created, we’ll pull in the required code for the backend.

**Note**: The final project for this tutorial can be found on [GitHub](https://github.com/howtographql/react-apollo). You can always use it as a reference whenever you get lost throughout the course of the following chapters. Also note that each code block is annotated with a filename. These annotations directly link to the corresponding file on GitHub so you can clearly see where to put the code and what the end result will look like.

## Frontend

#### Creating the app

The first step is to create a React project! As mentioned in the beginning, we’ll use create-react-app for that.

To create new project, run the command below:

yarn create react-app hackernews-react-apollo

**Note**: This tutorial uses [yarn](https://yarnpkg.com/) for dependency management. Find instructions for how you can install it [here](https://yarnpkg.com/en/docs/install). If you prefer using npm, you can just run the equivalent commands.

This will create a new directory called hackernews-react-apollo that has all the basic configuration setup.

Make sure everything works by navigating into the directory and starting the app:

**cd** hackernews-react-apollo

yarn start

This will open a browser and navigate to http://localhost:3000 where the app is running. If everything went well, we’ll see the following:

To improve the project structure, create two directories, both inside the src folder. The first is called components and will hold all our React components. Create a second directory called styles to hold all of the CSS for the project.

App.js is a component, so move it into components. You can also move App.test.js there as well (or delete it as we won’t use it here). App.css and index.css contain styles, so move them into styles. We also need to change the references to these files in both index.js and App.js accordingly:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

import React from 'react';

import ReactDOM from 'react-dom';

import './styles/index.css';

import App from './components/App';

import reportWebVitals from './reportWebVitals';

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

import logo from './../logo.svg';

import './../styles/App.css';

The project structure should now look as follows:

.

├── README.md

├── node\_modules

├── package.json

├── public

│ ├── favicon.ico

│ ├── index.html

│ └── manifest.json

│ └── logo192.png

│ └── logo512.png

│ └── robot.txt

├── src

│ ├── components

│ │ └── App.js

│ │ └── App.test.js

│ ├── index.js

│ ├── logo.svg

│ ├── reportWebVitals.js

│ ├── setupTests.js

│ └── styles

│ ├── App.css

│ └── index.css

└── yarn.lock

### Prepare Styling

This tutorial is about the concepts of GraphQL and how we can use it from within a React application, so we want to spend as little time as possible on styling. To reduce the usage of CSS in this project, we’ll use the [Tachyons](http://tachyons.io/) library which provides a number of CSS classes.

Open public/index.html and add a third link tag right below the two existing ones that pulls in Tachyons:

[**.../hackernews-react-apollo/public/index.html**](https://github.com/howtographql/react-apollo/blob/master/public/index.html)

<link rel="apple-touch-icon" href="%PUBLIC\_URL%/logo192.png" />

*<!--*

*manifest.json provides metadata used when your web app is installed on a*

*user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/*

*-->*

<link rel="manifest" href="%PUBLIC\_URL%/manifest.json" />

*<!--*

*Notice the use of %PUBLIC\_URL% in the tags above.*

*It will be replaced with the URL of the `public` folder during the build.*

*Only files inside the `public` folder can be referenced from the HTML.*

*Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC\_URL%/favicon.ico" will*

*work correctly both with client-side routing and a non-root public URL.*

*Learn how to configure a non-root public URL by running `npm run build`.*

*-->*

<link

rel="stylesheet"

href=<https://unpkg.com/tachyons@4.12.0/css/tachyons.min.css>

/>

Since we still want to have a bit more custom styling, we also prepared some styles that need to be included in the project.

Open index.css and replace its content with the following:

[**.../hackernews-react-apollo/src/styles/index.css**](https://github.com/howtographql/react-apollo/blob/master/src/styles/index.css)

body {

margin: 0;

padding: 0;

font-family: Verdana, Geneva, sans-serif;

}

input {

max-width: 500px;

}

.gray {

color: #828282;

}

.orange {

background-color: #ff6600;

}

.background-gray {

background-color: **rgb**(246, 246, 239);

}

.f11 {

font-size: 11px;

}

.w85 {

width: 85%;

}

.button {

font-family: monospace;

font-size: 10pt;

color: black;

background-color: buttonface;

text-align: center;

padding: 2px 6px 3px;

border-width: 2px;

border-style: outset;

border-color: buttonface;

cursor: pointer;

max-width: 250px;

}

### Install Apollo Client

Next, we need to pull in the functionality of Apollo Client (and its React hooks) which comes in several packages:

[**$.../hackernews-react-apollo**](https://github.com/howtographql/react-apollo/blob/master/hackernews-react-apollo)

yarn add @apollo/client graphql

Here’s an overview of the packages we installed:

* [@apollo/client](https://github.com/apollographql/apollo-client) contains all the pieces needed to wire up the GraphQL client for our app. It exposes the ApolloClient, a provider to wrap around the React app called ApolloProvider, custom hooks such as useQuery, and much more.
* [graphql](https://github.com/graphql/graphql-js) contains Facebook’s reference implementation of GraphQL - Apollo Client uses some of its functionality within.

That’s it, we’re ready to write some code! 🚀

### Configure ApolloClient

Apollo abstracts away all lower-level networking logic and provides a nice interface to the GraphQL server. In contrast to working with REST APIs, we don’t have to deal with constructing our own HTTP requests any more - instead we can simply write queries and mutations and send them using an ApolloClient instance.

The first thing we have to do when using Apollo is configure our ApolloClient instance. It needs to know the endpoint of our GraphQL API so it can deal with the network connections.

Open src/index.js and replace the contents with the following:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

import React from 'react';

import ReactDOM from 'react-dom';

import './styles/index.css';

import App from './components/App';

*// 1*

import {

ApolloProvider,

ApolloClient,

createHttpLink,

InMemoryCache

} from '@apollo/client';

*// 2*

const httpLink = **createHttpLink**({

uri: 'http://localhost:4000'

});

*// 3*

const client = new ApolloClient({

link: httpLink,

cache: new InMemoryCache()

});

*// 4*

ReactDOM.**render**(

<ApolloProvider client={client}>

<App />

</ApolloProvider>,

document.**getElementById**('root')

);

Let’s take a look at what’s going on in the code snippet above:

1. We import all the dependencies we need to wire up the Apollo client, all from @apollo/client.
2. We create the httpLink that will connect our ApolloClient instance with the GraphQL API. The GraphQL server will be running on http://localhost:4000.
3. We instantiate ApolloClient by passing in the httpLink and a new instance of an InMemoryCache.
4. Finally, we render the root component of our React app. The App is wrapped with the higher-order component ApolloProvider that gets passed the client as a prop.

That’s it, we’re all set to start for loading some data into our app! 😎

## Backend

### Downloading the Server Code

As mentioned above, for the backend in this tutorial we’ll simply use the final project from the [Node tutorial](https://www.howtographql.com/graphql-js/0-introduction).

In the terminal, navigate to the hackernews-react-apollo directory and run the following commands:

[**$.../hackernews-react-apollo**](https://github.com/howtographql/react-apollo/blob/master/hackernews-react-apollo)

curl https://codeload.github.com/howtographql/react-apollo/tar.gz/starter | **tar** -xz --strip=1 react-apollo-starter/server

**Note**: If you are on Windows, you may want to install [Git CLI](https://git-scm.com/) to avoid potential problems with commands such as curl.

We now have a new directory called server inside our project that contains all the code needed for our backend.

Before we start the server, let’s quickly understand the main components:

* prisma: This directory holds all the files that relate to our [Prisma](https://www.prisma.io/) setup. Prisma Client is used to access the database in our GraphQL resolvers (similar to an ORM).
  + schema.prisma defines our data model for the project. It uses the [Prisma Schema Language](https://www.prisma.io/docs/reference/tools-and-interfaces/prisma-schema) to define the shape of our databases tables and the relations between them.
  + dev.db is a SQLite database that will be used to store and retrieve data for this tutorial
* src: This directory holds the source files for our GraphQL server.
* schema.graphql contains our **application schema**. The application schema defines the GraphQL operations we can send from the frontend. We’ll take a closer look at this file in just a bit.
* resolvers contains the [resolver functions](https://www.prisma.io/blog/graphql-server-basics-the-schema-ac5e2950214e#resolvers-implement-the-api) for the operations defined in the application schema.
* index.js is the entry point for our GraphQL server.

From the mentioned files, only the application schema defined in server/src/schema.graphql is relevant for you as a frontend developer. This file contains the [GraphQL schema](https://www.prisma.io/blog/graphql-server-basics-the-schema-ac5e2950214e) which defines all the operations (queries, mutations and subscriptions) we can send from your frontend app.

Here is what it looks like:

[**.../react-apollo/blob/master/server/src/schema.graphql**](https://github.com/howtographql/react-apollo/blob/master/blob/master/server/src/schema.graphql)

type Query {

info: String!

feed(filter: String, skip: Int, take: Int, orderBy: LinkOrderByInput): Feed!

}

type Feed {

id: ID!

links: [Link!]!

count: Int!

}

type Mutation {

post(url: String!, description: String!): Link!

signup(email: String!, password: String!, name: String!): AuthPayload

login(email: String!, password: String!): AuthPayload

vote(linkId: ID!): Vote

}

type Subscription {

newLink: Link

newVote: Vote

}

type AuthPayload {

token: String

user: User

}

type User {

id: ID!

name: String!

email: String!

links: [Link!]!

}

type Link {

id: ID!

description: String!

url: String!

postedBy: User

votes: [Vote!]!

createdAt: DateTime!

}

type Vote {

id: ID!

link: Link!

user: User!

}

input LinkOrderByInput {

description: Sort

url: Sort

createdAt: Sort

}

enum Sort {

asc

desc

}

scalar DateTime

This schema allows for the following operations:

* Queries:
  + feed: Retrieves all links from the backend, note that this query also allows for filter, sorting and pagination arguments
* Mutations:
  + post: Allows authenticated users to create a new link
  + signup: Create an account for a new user
  + login: Login an existing user
  + vote: Allows authenticated users to vote for an existing link
* Subscriptions:
  + newLink: Receive realtime updates when a new link is created
  + newVote: Receive realtime updates when a vote was submitted

For example, we can send the following feed query to retrieve the first 10 links from the server:

{

feed(skip: 0, take: 10) {

links {

description

url

postedBy {

name

}

}

}

}

Or the signup mutation to create a new user:

mutation {

signup(name: "Sarah", email: "sarah@prisma.io", password: "graphql") {

token

user {

id

}

}

}

### Creating a Database and Generating Prisma Client

There is one thing left to do before we can start our server and begin sending queries and mutations to it. We need a database and a generated Prisma Client so that we can actually store and retrieve data.

Prisma [supports several relational databases](https://www.prisma.io/docs/more/supported-databases), including Postgres, MySQL, and SQLite.

For this tutorial, we’ll keep things simple and use SQLite. It’s a filesystem database that is very easy to get started with. It should be noted, however, that SQLite may not be suitable for production purposes.

There is a file called dev.db located in the server/prisma directory. This file is our SQLite database.

**Note**: You are free to use Postgres or MySQL for this tutorial if you prefer. All aspects of the tutorial will still work with those databases.

Next, let’s run database migrations and generate Prisma Client.

We need to change directories into server and run some commands to generate Prisma Client.

Before doing so, make sure to install the dependencies.

**cd** server

yarn

[**.../react-apollo/server**](https://github.com/howtographql/react-apollo/blob/master/server)

npx prisma generate

### Exploring the server

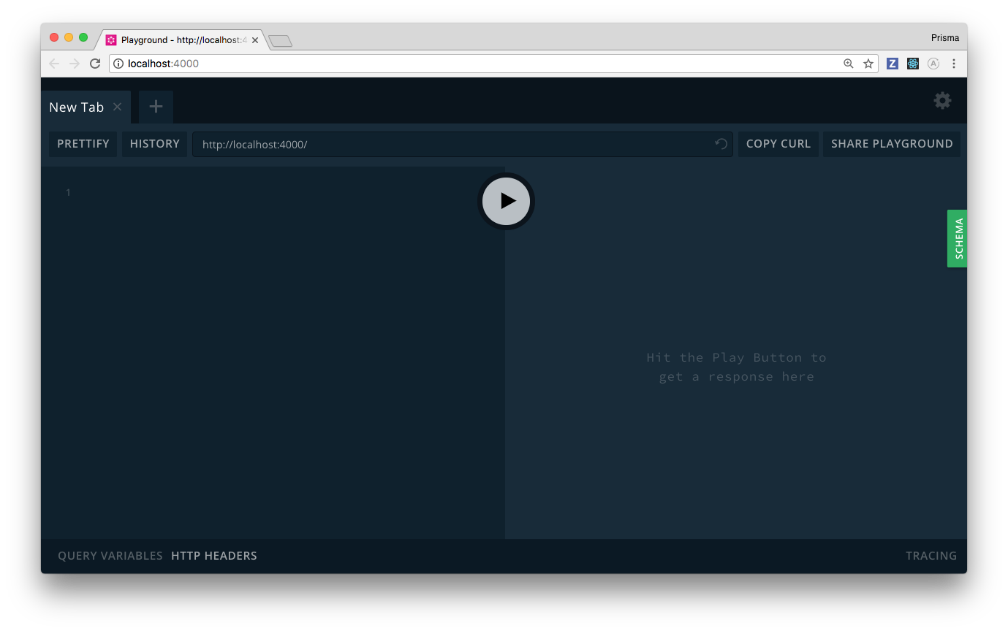
With Prisma Client generated, we can now explore our server.

Navigate into the server directory and run the following commands to start the server:

[**$.../hackernews-react-apollo/server**](https://github.com/howtographql/react-apollo/blob/master/server)

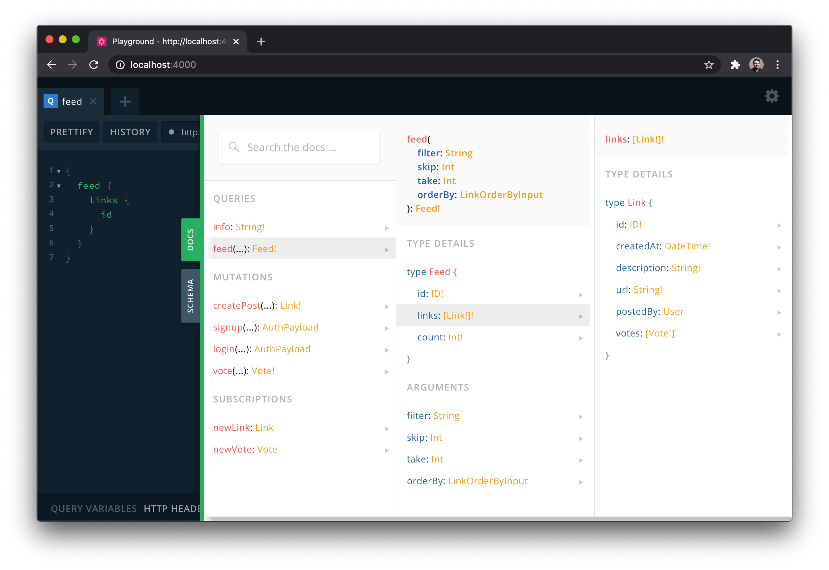
yarn dev

The yarn dev executes the dev script defined in package.json. The script first starts the server using [nodemon](https://www.npmjs.com/package/nodemon) (which is then running on http://localhost:4000) and then opens up a [GraphQL Playground](https://github.com/graphql/graphql-playground) for us to explore and work with the API.



A Playground is a “GraphQL IDE”, providing an interactive environment that allows to send queries, mutations and subscriptions to your GraphQL API. It is similar to a tool like [Postman](https://www.getpostman.com/) which you might know from working with REST APIs, but comes with a lot of additional benefits.

The first thing to note about the Playground is that it has built-in documentation for its GraphQL API. This documentation is generated based on the GraphQL schema and can be opened by clicking the green **SCHEMA** button on the right edge of the Playground. Consequently, it shows you the same information you saw in the application schema above:



The left pane of the Playground is the editor that you can use to write your queries, mutations and subscriptions. Once you click the play button in the middle, your request is sent and the server’s response will be displayed in the results pane on the right.

Copy the following two mutations into the editor pane.

mutation CreatePrismaLink {

post(

description: "Prisma gives you a powerful database toolkit 😎"

url: "https://prisma.io"

) {

id

}

}

mutation CreateApolloLink {

post(

description: "The best GraphQL client for React"

url: "https://www.apollographql.com/docs/react/"

) {

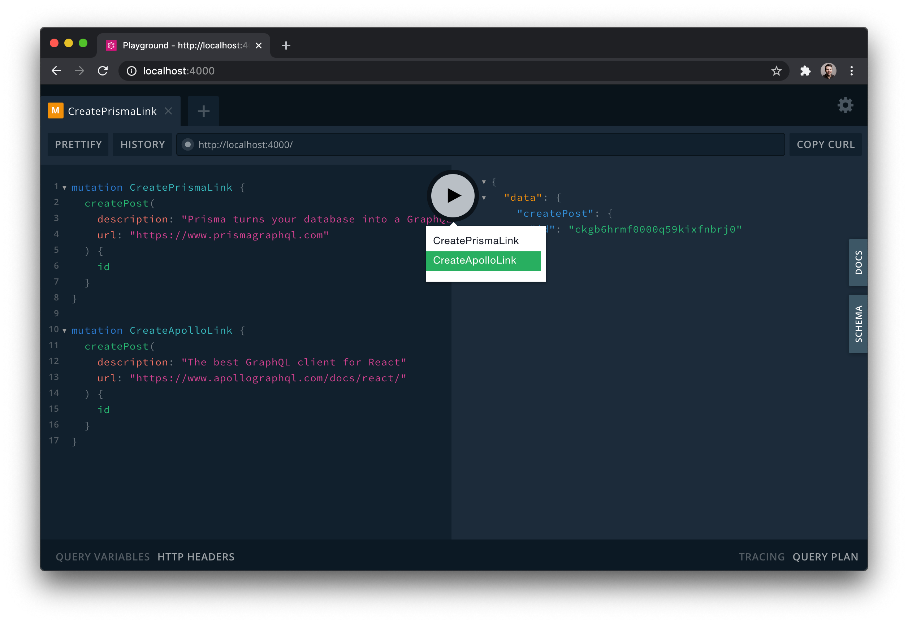
id

}

}

Since you’re adding two mutations to the editor at once, the mutations need to have operation names. In your case, these are CreatePrismaLink and CreateApolloLink.

Click the **Play** button in the middle of the two panes and select each mutation from the dropdown exactly once.



This creates two new Link records in the database. You can verify that the mutations actually worked by sending the following query in the already open Playground:

{

feed {

links {

id

description

url

}

}

}

If everything went well, the query will return the following data (the ids will of course be different in your case since they were generated by Prisma and are globally unique):

{

"data": {

"feed": {

"links": [

{

"id": "43f7d9a2-586e-40e7-8e84-d73d3abe543f",

"description": "Prisma gives you a powerful database toolkit 😎",

"url": "https://prisma.io"

},

{

"id": "8791ed40-f685-4f8f-8e26-638af8074973",

"description": "The best GraphQL client for React",

"url": "https://www.apollographql.com/docs/react/"

}

]

}

}

}

Fantastic, our server works! 👏

# React + Apollo – Queries: Loading Links

<https://www.howtographql.com/react-apollo/2-queries-loading-links/>

## Preparing the React components

The first piece of functionality we’ll implement in the app is loading and displaying a list of Link elements. We’ll walk up our way in the React component hierarchy and start with the component that will render a single link.

Create a new file called Link.js in the components directory and add the following code:

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

import React from 'react';

const Link = (props) => {

const { link } = props;

return (

<div>

<div>

{link.description} ({link.url})

</div>

</div>

);

};

export default Link;

This is a simple React component that expects a link in its props and renders the link’s description and url. Easy as pie! 🍰

Next, let’s implement the component that renders a list of links.

Again, in the components directory, go ahead and create a new file called LinkList.js. Then add the following code:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

import React from 'react';

import Link from './Link';

const LinkList = () => {

const linksToRender = [

{

id: 'link-id-1',

description:

'Prisma gives you a powerful database toolkit 😎',

url: 'https://prisma.io'

},

{

id: 'link-id-2',

description: 'The best GraphQL client',

url: 'https://www.apollographql.com/docs/react/'

}

];

return (

<div>

{linksToRender.**map**((link) => (

<Link key={link.id} link={link} />

))}

</div>

);

};

export default LinkList;

Here, we’re using local mock data for now to make sure the component setup works. We’ll soon replace this with some actual data loaded from the server - patience, young Padawan!

To complete the setup, open App.js and replace the current contents with the following:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

import React, { Component } from 'react';

import LinkList from './LinkList';

class App extends Component {

**render**() {

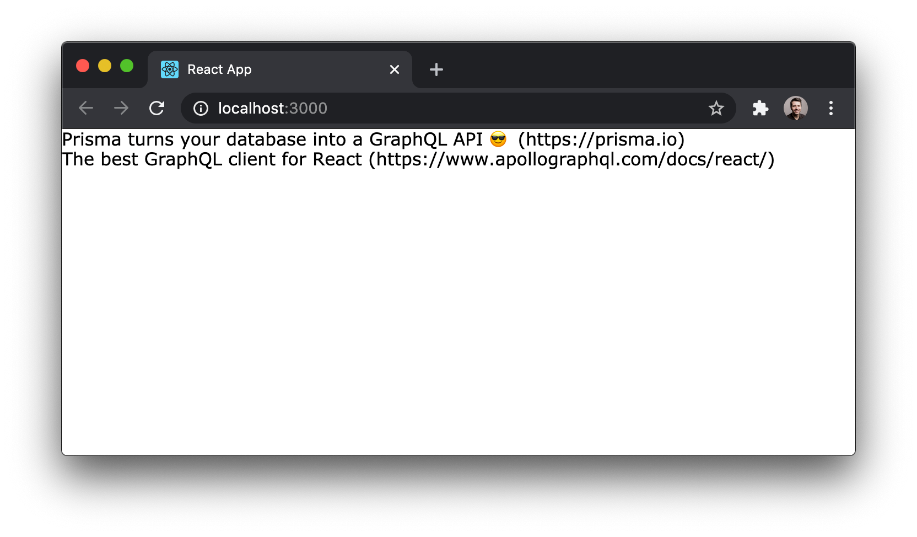
return <LinkList />;

}

}

export default App;

Run the app to check if everything works so far! The app should now display the two links from the linksToRender array:



## Writing the GraphQL query

Let’s now load the actual links that are stored in the database. The first thing we need to do for that is define the GraphQL query we want to send to the API.

Here is what it looks like:

{

feed {

id

links {

id

createdAt

description

url

}

}

}

We could now simply execute this query in a [Playground](https://github.com/graphql/graphql-playground) (against the application schema) and retrieve the results from the GraphQL server. But how can we use it inside our JavaScript code?

## Queries with Apollo Client

The most common (and probably the most flexible) way of making queries with Apollo Client is to use the useQuery hook it provides. With the useQuery hook, all we need to do is pass a GraphQL query document in and Apollo will take care of the fetching and will surface the returned data and any errors for us.

Open up LinkList.js and add the query to the top of the file:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

import React from 'react';

import Link from './Link';

import { useQuery, gql } from '@apollo/client';

const FEED\_QUERY = gql`

{

feed {

id

links {

id

createdAt

url

description

} }

}

`;

Next, instead of iterating over the hardcoded linksToRender array, pass the FEED\_QUERY query document into useQuery and iterate over the returned links.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const LinkList = () => {

const { data } = **useQuery**(FEED\_QUERY);

return (

<div>

{data && (

<>

{data.feed.links.**map**((link) => (

<Link key={link.id} link={link} />

))}

</>

)}

</div>

);

};

Let’s take a moment to walk through what’s happening with this new code.

The FEED\_QUERY variable uses gql, a library that uses [tagged template literals](https://wesbos.com/tagged-template-literals) to parse the GraphQL query document we define. This query document is then passed into the useQuery hook in the LinkList component.

This hook returns three items that are relevant for our purposes at this point:

1. loading: Is true as long as the request is still ongoing and the response hasn’t been received.
2. error: In case the request fails, this field will contain information about what exactly went wrong.
3. data: This is the actual data that was received from the server. It has the links property which represents a list of Link elements.

The injected props actually contain even more functionality. You can read more in the [API overview](https://www.apollographql.com/docs/react/essentials/queries.html#render-prop).

When the LinkList component initially renders, there won’t be any information on the data variable. For this reason, we need to check that data is truthy before trying to render any of the links that will come out of it. Once our GraphQL request resolves with some data, the LinkList component will re-render and data will be truthy. Our links are available on data.feed.links which we can map over to render.

That’s it! You should see the exact same screen as before.

**Note**: If the browser on http://localhost:4000 only says error and is empty otherwise, you probably forgot to have your server running. Note that for the app to work the server needs to run as well - so you have two running processes in your terminal: One for the server and one for the React app. To start the server, navigate into the server directory and run yarn start.

# React + Apollo – Mutations: Creating Links

<https://www.howtographql.com/react-apollo/3-mutations-creating-links/>

There are two top-level operation types in GraphQL: queries and mutations.

Queries are used when we want to read data.

Mutations are used when we want to change data. We use mutations when we want to create, update, or delete records.

In this section, we’ll learn how to send mutations with Apollo. Doing so is actually not that much different than sending queries and follows similar steps.

1. Write the mutation as a JavaScript constant using the gql parser function
2. Use the useMutation hook provided by Apollo Client to send mutations to our GraphQL server
3. Read the result of the mutation

## Preparing the React components

Like before, let’s start by writing the React component where users will be able to add new links.

Create a new file in the src/components directory and call it CreateLink.js. Then paste the following code into it:

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

import React, { useState } from 'react';

const CreateLink = () => {

const [formState, setFormState] = **useState**({

description: '',

url: ''

});

return (

<div>

<form

onSubmit={(e) => {

e.**preventDefault**();

}}

>

<div className="flex flex-column mt3">

<input

className="mb2"

value={formState.description}

onChange={(e) =>

**setFormState**({

...formState,

description: e.target.value

})

}

type="text"

placeholder="A description for the link"

/>

<input

className="mb2"

value={formState.url}

onChange={(e) =>

**setFormState**({

...formState,

url: e.target.value

})

}

type="text"

placeholder="The URL for the link"

/>

</div>

<button type="submit">Submit</button>

</form>

</div>

);

};

export default CreateLink;

This is a standard setup for a React component with two input fields where users can provide the url and description of the Link they want to create. The data that’s typed into these fields is held in the component’s local state by way of the useState hook.

## Writing the mutation

Our job now is to take the user input and send it as arguments in a GraphQL mutation.

First, we need to define the mutation in our JavaScript code and use the useMutation hook to fire the mutation.

In CreateLink.js, add the following statement to the top of the file:

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

import { useMutation, gql } from '@apollo/client';

const CREATE\_LINK\_MUTATION = gql`

mutation PostMutation(

$description: String!

$url: String!

) {

post(description: $description, url: $url) {

id

createdAt

url

description

}

}

`;

Next, pass the CREATE\_LINK\_MUTATION to the useMutation hook and pass in the data provided in the input fields as variables.

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

const CreateLink = () => {

const [formState, setFormState] = **useState**({

description: '',

url: ''

});

const [createLink] = **useMutation**(CREATE\_LINK\_MUTATION, {

variables: {

description: formState.description,

url: formState.url

}

});

*// ...*

};

When we use the useMutation hook, we need to destructure out a function that can be used to call the mutation. That’s what createLink is in the code block above. We’re now free to call the function whenever we need to when the component renders.

Make a call to createLink in the onSubmit event on the form tag.

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

return (

<div>

<form

onSubmit={(e) => {

e.**preventDefault**();

**createLink**();

}}

>

...

</form>

</div>

);

We’re now ready to check wether the mutations are working.

Open App.js and change it up as follows:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

import React from 'react';

import CreateLink from './CreateLink';

import LinkList from './LinkList';

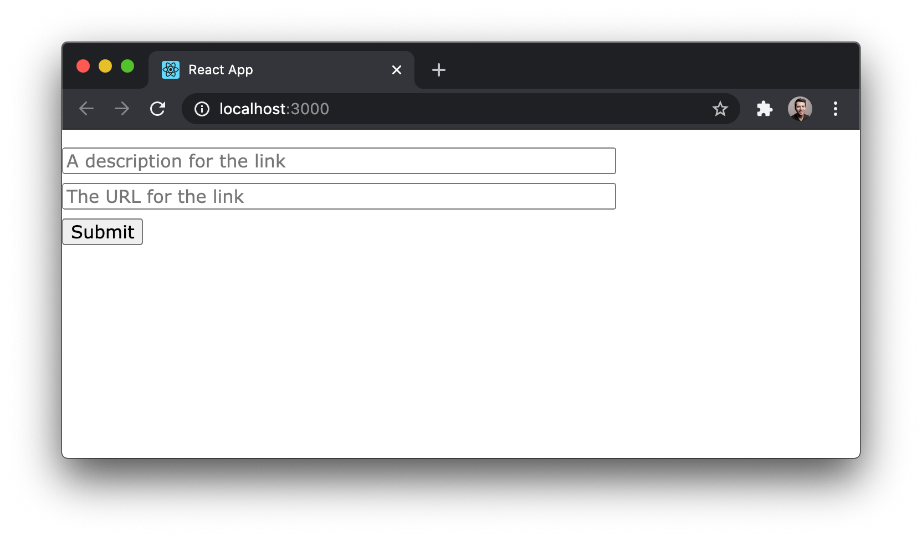
const App = () => {

return <CreateLink />;

};

export default App;

Now, when we run yarn start, we should see the following:



Two input fields and a **Submit** button - not very pretty but functional.

Enter some data into the fields, e.g.:

* **Description**: The best learning resource for GraphQL
* **URL**: howtographql.com

Then click the **Submit** button. You won’t get any visual feedback in the UI, but let’s see if the query actually worked by checking the current list of links in a Playground.

You can open a Playground again by navigating to http://localhost:4000 in your browser. Then send the following query:

*# Try to write the query here*

{

feed {

links {

description

url

}

}

}

You’ll see the following server response:

{

"data": {

"feed": {

"links": [

*// ...*

{

"description": "The best learning resource for GraphQL",

"url": "howtographql.com"

}

]

}

}

}

Awesome! The mutation works, great job! 💪

# React + Apollo – Routing

<https://www.howtographql.com/react-apollo/4-routing/>

In this section, we’ll see how to use the [React Router](https://github.com/ReactTraining/react-router) with Apollo to implement navigation!

Install dependencies

Let’s start by adding the dependencies we’ll need.

[**$.../hackernews-react-apollo**](https://github.com/howtographql/react-apollo/blob/master/hackernews-react-apollo)

yarn add react-router-dom@^6.2.1

## Create a Header

Before moving on to configure the different routes for the app, we need to create a Header component that will hold the navigation links.

Create a new file in src/components and call it Header.js. Then paste the following code inside of it:

[**.../hackernews-react-apollo/src/components/Header.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Header.js)

import React from 'react';

import { Link } from 'react-router-dom';

const Header = () => {

return (

<div className="flex pa1 justify-between nowrap orange">

<div className="flex flex-fixed black">

<Link to="/" className="no-underline black">

<div className="fw7 mr1">Hacker News</div>

</Link>

<Link to="/" className="ml1 no-underline black">

new

</Link>

<div className="ml1">|</div>

<Link

to="/create"

className="ml1 no-underline black"

>

submit

</Link>

</div>

</div>

);

};

export default Header;

The Header component currently just renders two Link components that can be used to navigate between the LinkList and the CreateLink components.

Don’t get confused by the “other” Link component that is used here. The one that you’re using in the Header has nothing to do with the Link component that you wrote before, they just happen to have the same name. This [Link](https://github.com/ReactTraining/react-router/blob/master/packages/react-router-dom/docs/api/Link.md) stems from the react-router-dom package and allows us to navigate between routes inside of your application.

## Setup routes

Let’s configure the different routes for the app in the project’s root component: App.

Open up App.js and update it to include the Header as well as LinkList and the CreateLink components under different routes:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

import React from 'react';

import CreateLink from './CreateLink';

import Header from './Header';

import LinkList from './LinkList';

import { Route, Routes } from 'react-router-dom';

const App = () => {

return (

<div className="center w85">

<Header />

<div className="ph3 pv1 background-gray">

<Routes>

<Route path="/" element={<LinkList/>} />

<Route

path="/create"

element={<CreateLink/>}

/>

</Routes>

</div>

</div>

);

};

export default App;

We now need to wrap the App with BrowserRouter so that all child components of App will get access to the routing functionality.

Open index.js and add the following import statement to the top:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

import { BrowserRouter } from 'react-router-dom';

Now update ReactDOM.render and wrap the whole app with BrowserRouter:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

ReactDOM.**render**(

<BrowserRouter>

<ApolloProvider client={client}>

<App />

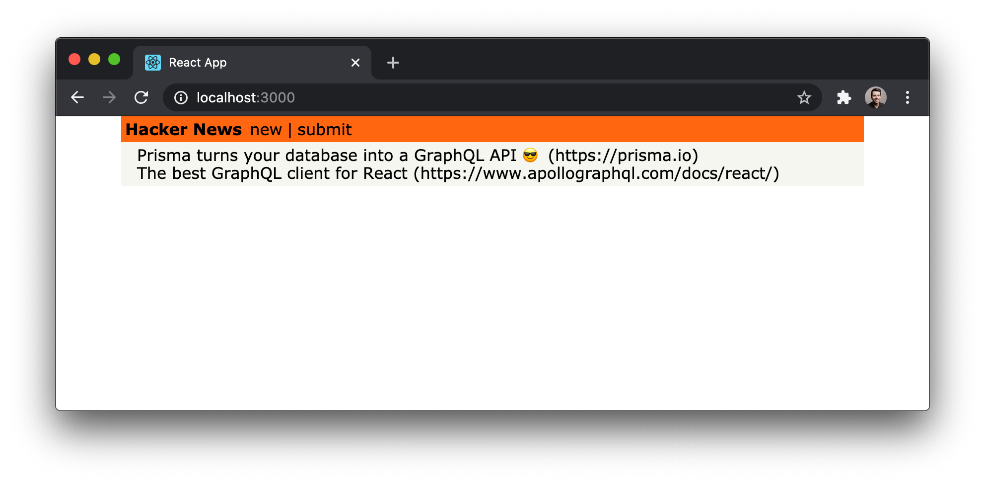
</ApolloProvider>

</BrowserRouter>,

document.**getElementById**('root')

);

If we run the app again, we can now access two URLs. http://localhost:3000/ will render LinkList and http://localhost:3000/create renders the CreateLink component we created in the previous section.



## Implement Navigation

To wrap up this section, we need to implement an automatic redirect from the CreateLink component to the LinkList component after a mutation is performed. To do this, we can use the onCompleted function that is provided by Apollo when mutations are performed.

Open CreateLink.js and update it to include the useNavigate hook from React Router. In the body of the function, create a navigate reference and use it within the onCompleted callback. This callback runs after the mutation is completed.

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

*// ...*

import { useNavigate } from 'react-router-dom';

const CreateLink = () => {

const navigate = **useNavigate**();

const [createLink] = **useMutation**(CREATE\_LINK\_MUTATION, {

variables: {

description: formState.description,

url: formState.url

},

onCompleted: () => **navigate**('/')

});

*// ...*

};

After the mutation completes, React Router will navigate back to the LinkList component that’s accessible on the root route: /.

**Note**: With our current setup, we won’t see the newly created Link, we’ll just redirect to the main route. We could refresh the page to see the changes made. We’ll see how to update the data after the mutation completes in the More Mutations and Updating the Store chapter!

# React + Apollo – Authentication

<https://www.howtographql.com/react-apollo/5-authentication/>

In this section, we’ll see how to implement authentication with Apollo to provide signup and login features in our app.

## Prepare the React Components

As in the sections before, we’ll set the stage for the login functionality by preparing the React components that are needed for this feature. We’ll start by building the Login component.

Create a new file in src/components and call it Login.js. Then paste the following code into it:

[**.../hackernews-react-apollo/src/components/Login.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Login.js)

import React, { useState } from 'react';

import { useNavigate } from 'react-router-dom';

const Login = () => {

const navigate = **useNavigate**();

const [formState, setFormState] = **useState**({

login: true,

email: '',

password: '',

name: ''

});

return (

<div>

<h4 className="mv3">

{formState.login ? 'Login' : 'Sign Up'}

</h4>

<div className="flex flex-column">

{!formState.login && (

<input

value={formState.name}

onChange={(e) =>

**setFormState**({

...formState,

name: e.target.value

})

}

type="text"

placeholder="Your name"

/>

)}

<input

value={formState.email}

onChange={(e) =>

**setFormState**({

...formState,

email: e.target.value

})

}

type="text"

placeholder="Your email address"

/>

<input

value={formState.password}

onChange={(e) =>

**setFormState**({

...formState,

password: e.target.value

})

}

type="password"

placeholder="Choose a safe password"

/>

</div>

<div className="flex mt3">

<button

className="pointer mr2 button"

onClick={() => console.**log**('onClick')}

>

{formState.login ? 'login' : 'create account'}

</button>

<button

className="pointer button"

onClick={(e) =>

**setFormState**({

...formState,

login: !formState.login

})

}

>

{formState.login

? 'need to create an account?'

: 'already have an account?'}

</button>

</div>

</div>

);

};

export default Login;

Let’s quickly understand the structure of this new component, which can have two major states:

* One state is **for users that already have an account** and only need to login. In this state, the component will only render two input fields for the user to provide their email and password. Notice that formState.login will be true in this case.
* The second state is for **users that haven’t created an account yet**, and thus still need to sign up. Here, we render a third input field where users can provide their name. In this case, formState.login will be false.

In the onClick handler in the submit button, we’ll eventually call the appropriate mutations for these two actions.

Next, we also need to provide the constants.js file that we use to define the key for the credentials that we’re storing in the browser’s localStorage.

**Warning**: Storing JWTs in localStorage is not a safe approach to implement authentication on the frontend. Because this tutorial is focused on GraphQL, we want to keep things simple and therefore are using it here. You can read more about this topic [here](https://www.rdegges.com/2018/please-stop-using-local-storage/).

In src, create a new file called constants.js and add the following definition:

[**.../hackernews-react-apollo/src/constants.js**](https://github.com/howtographql/react-apollo/blob/master/src/constants.js)

export const AUTH\_TOKEN = 'auth-token';

With that component in place, we can add a new route to our routing setup.

Open App.js and update it to include the new route:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

*// ...*

import Login from './Login';

const App = () => {

return (

<div className="center w85">

<Header />

<div className="ph3 pv1 background-gray">

<Routes>

<Route path="/" element={<LinkList/>} />

<Route

path="/create"

element={<CreateLink/>}

/>

<Route path="/login" element={<Login/>} />

</Routes>

</div>

</div>

);

};

export default App;

Add a Link to the Header to allow users to navigate to the Login page.

Open Header.js and update render to look as follows:

[**.../hackernews-react-apollo/src/components/Header.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Header.js)

import React from 'react';

import { Link, useNavigate } from 'react-router-dom';

import { AUTH\_TOKEN } from '../constants';

const Header = () => {

const navigate = **useNavigate**();

const authToken = localStorage.**getItem**(AUTH\_TOKEN);

return (

<div className="flex pa1 justify-between nowrap orange">

<div className="flex flex-fixed black">

<Link to="/" className="no-underline black">

<div className="fw7 mr1">Hacker News</div>

</Link>

<Link to="/" className="ml1 no-underline black">

new

</Link>

<div className="ml1">|</div>

<Link

to="/search"

className="ml1 no-underline black"

>

search

</Link>

{authToken && (

<div className="flex">

<div className="ml1">|</div>

<Link

to="/create"

className="ml1 no-underline black"

>

submit

</Link>

</div>

)}

</div>

<div className="flex flex-fixed">

{authToken ? (

<div

className="ml1 pointer black"

onClick={() => {

localStorage.**removeItem**(AUTH\_TOKEN);

**navigate**(`/`);

}}

>

logout

</div>

) : (

<Link

to="/login"

className="ml1 no-underline black"

>

login

</Link>

)}

</div>

</div>

);

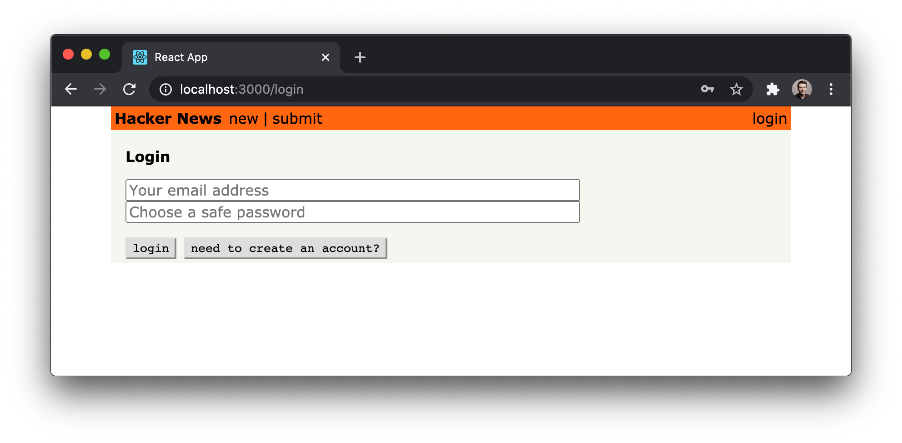
};

export default Header;

We first retrieve the authToken from local storage. If the authToken is not available, the **submit** button won’t be rendered. This way, we can make sure only authenticated users can create new links.

We’re also adding a second button to the right of the Header that users can use to login and logout.

Here is what the ready component looks like:



Perfect, we’re all set to implement the authentication functionality.

## Using the authentication mutations

signup and login are two regular GraphQL mutations we can use in the same way as we did with the createLink mutation from before.

Open Login.js and add the following two definitions to the top of the file:

[**.../hackernews-react-apollo/src/components/Login.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Login.js)

const SIGNUP\_MUTATION = gql`

mutation SignupMutation(

$email: String!

$password: String!

$name: String!

) {

signup(

email: $email

password: $password

name: $name

) {

token

}

}

`;

const LOGIN\_MUTATION = gql`

mutation LoginMutation(

$email: String!

$password: String!

) {

login(email: $email, password: $password) {

token

}

}

`;

Both mutations look very similar to the mutations we’ve already seen. They take a number of arguments and return the token that we can attach to subsequent requests to authenticate the user (i.e. indicate that a request is made on behalf of that user). You’ll learn 🔜 how to do so.

Next, find the div element that has the class names flex mt3 and replace it with the following:

[**.../hackernews-react-apollo/src/components/Login.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Login.js)

<div className="flex mt3">

<button

className="pointer mr2 button"

onClick={formState.login ? login : signup}

>

{formState.login ? 'login' : 'create account'}

</button>

<button

className="pointer button"

onClick={(e) =>

**setFormState**({

...formState,

login: !formState.login

})

}

>

{formState.login

? 'need to create an account?'

: 'already have an account?'}

</button>

</div>

The onClick event on the “login”/“create account” button uses a ternary to call one of two functions: login or signup. As the names imply, these functions will run mutations to log the user in or sign them up for a new account. Let’s put in the useMutation hook to make these actions happen.

[**.../hackernews-react-apollo/src/components/Login.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Login.js)

const [login] = **useMutation**(LOGIN\_MUTATION, {

variables: {

email: formState.email,

password: formState.password

},

onCompleted: ({ login }) => {

localStorage.**setItem**(AUTH\_TOKEN, login.token);

**navigate**('/');

}

});

const [signup] = **useMutation**(SIGNUP\_MUTATION, {

variables: {

name: formState.name,

email: formState.email,

password: formState.password

},

onCompleted: ({ signup }) => {

localStorage.**setItem**(AUTH\_TOKEN, signup.token);

**navigate**('/');

}

});

These two mutations use the useMutation hook from Apollo. They accept the GraphQL mutation documents we defined earlier and accept variables from the form. The onCompleted callback sets the user’s token in local storage and redirects them to the home page afterward.

Still in Login.js, add the following imports to the top of the file:

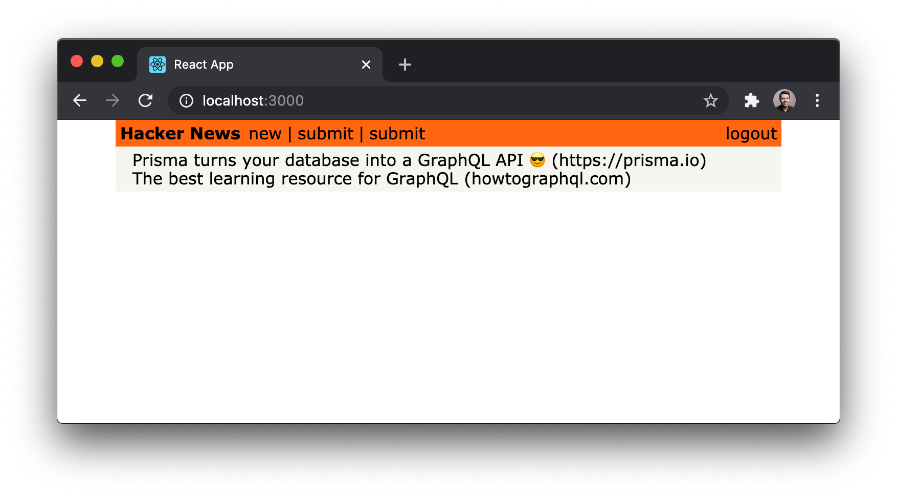
[**.../hackernews-react-apollo/src/components/Login.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Login.js)

import { useMutation, gql } from '@apollo/client';

import { useNavigate } from 'react-router-dom';

import { AUTH\_TOKEN } from '../constants';

We can now create an account by providing a name, email and password. Once we do that, the **submit** button will be rendered again:



If you haven’t done so before, go ahead and test the login functionality. Run yarn start and open http://localhost:3000/login. Then click the **need to create an account?** button and provide some user data for the user you’re creating. Finally, click the **create account** button. If all went well, the app navigates back to the home route and the user was created. We can verify that the new user is there by sending the users query in the **dev** Playground in the **database** project.

## Configuring Apollo with the authentication token

Now that users are able to log in and obtain a token that authenticates them against the GraphQL server, we need to make sure that the token gets attached to all requests that are sent to the API.

Since all the API requests are actually created and sent by the ApolloClient instance at the root of our app, we need to make sure it knows about the user’s token! Luckily, Apollo provides a nice way for authenticating all requests by using the concept of [middleware](http://dev.apollodata.com/react/auth.html#Header), implemented as an [Apollo Link](https://github.com/apollographql/apollo-link).

Open index.js and put the following code between the creation of the httpLink and the instantiation of ApolloClient:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

const authLink = **setContext**((\_, { headers }) => {

const token = localStorage.**getItem**(AUTH\_TOKEN);

return {

headers: {

...headers,

authorization: token ? `Bearer ${token}` : ''

}

};

});

Before moving on, we need to import the Apollo dependencies. Add the following to the top of index.js:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

import { setContext } from '@apollo/client/link/context';

This middleware will be invoked every time ApolloClient sends a request to the server. Apollo Links allow us to create middlewares that modify requests before they are sent to the server.

Let’s see how it works in our code: first, we get the authentication token from localStorage if it exists; after that, we return the headers to the context so httpLink can read them.

**Note**: You can read more about Apollo’s authentication [here](https://www.apollographql.com/docs/react/networking/authentication/).

We also need to make sure ApolloClient gets instantiated with the correct link - update the constructor call as follows:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

const client = new ApolloClient({

link: authLink.**concat**(httpLink),

cache: new InMemoryCache()

});

Then directly import the key we need to retrieve the token from localStorage on top of the same file:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

import { AUTH\_TOKEN } from './constants';

That’s it - now all our API requests will be authenticated if a token is available.

## Requiring authentication on the server-side

The last thing we might do in this chapter is check how to ensure only authenticated users are able to post new links. Plus, every Link that’s created by a post mutation should automatically set the User who sent the request for its postedBy field.

In our case, we’re allowing Links without an associated User to be submitted. This is for the sake of demonstration and may not be what you want for your own application. We can get a sense of how this works if we look at the server code in Mutation.js.

Open /server/src/resolvers/Mutation.js and give a look how it was implemented:

[**.../hackernews-react-apollo/server/src/resolvers/Mutation.js**](https://github.com/howtographql/react-apollo/blob/master/server/src/resolvers/Mutation.js)

async function **post**(parent, args, context, info) {

const { userId } = context;

let postedBy = undefined

if (userId) {

postedBy = { connect: { id: userId } }

}

const newLink = await context.prisma.link.**create**({

data: {

url: args.url,

description: args.description,

postedBy

}

});

context.pubsub.**publish**('NEW\_LINK', newLink); *// not important for now*

return newLink;

}

In this code block, we’re extracting the userId from the context object of the request and using it to directly [connect](https://www.prisma.io/docs/reference/tools-and-interfaces/prisma-client/relation-queries) it with the Link that’s created using the postedBy relation. The userId is placed on context by extracting it from the Authorization header when we set up the server context in index.js.

When creating posts without logging in, the Authorization header is not provided, then userId will be absent. In this case, postedBy will also be undefined and will be ignored by Prisma.

# React + Apollo – More Mutations and Updating the Store

<https://www.howtographql.com/react-apollo/6-more-mutations-and-updating-the-store/>

The next piece of functionality we’ll implement is the voting feature! Authenticated users are allowed to submit a vote for a link. The most upvoted links will later be displayed on a separate route!

## Preparing the React Components

Once more, the first step to implement this new feature is to make our React components ready for the expected functionality.

Open Link.js and update the returned JSX to look like this:

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

import { AUTH\_TOKEN } from '../constants';

*// ...*

const Link = (props) => {

const { link } = props;

const authToken = localStorage.**getItem**(AUTH\_TOKEN);

return (

<div className="flex mt2 items-start">

<div className="flex items-center">

<span className="gray">{props.index + 1}.</span>

{authToken && (

<div

className="ml1 gray f11"

style={{ cursor: 'pointer' }}

onClick={() => {console.**log**("Clicked vote button")}}

>

▲

</div>

)}

</div>

<div className="ml1">

<div>

{link.description} ({link.url})

</div>

{(

<div className="f6 lh-copy gray">

{link.votes.length} votes | by{' '}

{link.postedBy ? link.postedBy.name : 'Unknown'}{' '}

{**timeDifferenceForDate**(link.createdAt)}

</div>

)}

</div>

</div>

);

};

export default Link;

We’re already preparing the Link component to render the number of votes for each link and the name of the user that posted it. We’ll also render the upvote button if a user is currently logged in - that’s what we’re using the authToken for. If the Link is not associated with a User, the user’s name will be displayed as Unknown.

Notice that we’re also using a function called timeDifferenceForDate that gets passed the createdAt information for each link. The function will take the timestamp and convert it to a string that’s more user friendly, e.g. "3 hours ago".

Go ahead and implement the timeDifferenceForDate function next so we can import and use it in the Link component.

Create a new file called utils.js in the src directory and paste the following code into it:

[**.../hackernews-react-apollo/src/utils.js**](https://github.com/howtographql/react-apollo/blob/master/src/utils.js)

function **timeDifference**(current, previous) {

const milliSecondsPerMinute = 60 \* 1000;

const milliSecondsPerHour = milliSecondsPerMinute \* 60;

const milliSecondsPerDay = milliSecondsPerHour \* 24;

const milliSecondsPerMonth = milliSecondsPerDay \* 30;

const milliSecondsPerYear = milliSecondsPerDay \* 365;

const elapsed = current - previous;

if (elapsed < milliSecondsPerMinute / 3) {

return 'just now';

}

if (elapsed < milliSecondsPerMinute) {

return 'less than 1 min ago';

} else if (elapsed < milliSecondsPerHour) {

return (

Math.**round**(elapsed / milliSecondsPerMinute) +

' min ago'

);

} else if (elapsed < milliSecondsPerDay) {

return (

Math.**round**(elapsed / milliSecondsPerHour) + ' h ago'

);

} else if (elapsed < milliSecondsPerMonth) {

return (

Math.**round**(elapsed / milliSecondsPerDay) + ' days ago'

);

} else if (elapsed < milliSecondsPerYear) {

return (

Math.**round**(elapsed / milliSecondsPerMonth) + ' mo ago'

);

} else {

return (

Math.**round**(elapsed / milliSecondsPerYear) +

' years ago'

);

}

}

export function **timeDifferenceForDate**(date) {

const now = new Date().**getTime**();

const updated = new Date(date).**getTime**();

return **timeDifference**(now, updated);

}

Back in Link.js, import AUTH\_TOKEN and timeDifferenceForDate on top the file:

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

import { AUTH\_TOKEN } from '../constants';

import { timeDifferenceForDate } from '../utils';

Finally, each Link element will also render its position inside the list, so we have to pass down an index from the LinkList component.

Open LinkList.js and update the rendering of the Link component to include the index.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

return (

<div>

{data && (

<>

{data.feed.links.**map**((link, index) => (

<Link key={link.id} link={link} index={index} />

))}

</>

)}

</div>

);

Notice that the app won’t run at the moment since the votes are not yet included in the query. We’ll fix that next!

Open LinkList.js and update the definition of FEED\_QUERY to include votes. We should also export this query so that it can be imported in other files.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

export const FEED\_QUERY = gql`

{

feed {

id

links {

id

createdAt

url

description

postedBy {

id

name

}

votes {

id

user {

id

}

}

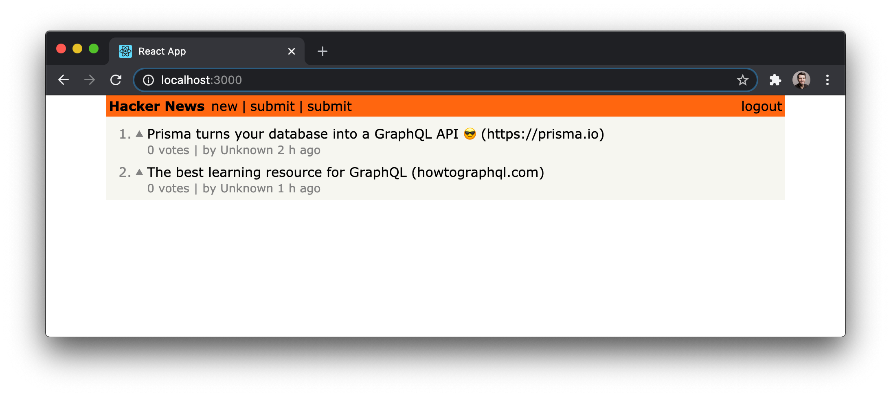
}

}

}

`;

Here we are including information about the user who posted a link as well as information about the links’ votes in the query’s payload. We can now run the app again and the links will be properly displayed.



**Note**: If you’re not able to fetch the Links, restart the server and reload the browser. You could also check if everything is working as expected on GraphQL Playground!

Let’s now move on and implement the vote mutation!

## Calling the Mutation

Open Link.js and add the following mutation definition to the top of the file.

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

const VOTE\_MUTATION = gql`

mutation VoteMutation($linkId: ID!) {

vote(linkId: $linkId) {

id

link {

id

votes {

id

user {

id

}

}

}

user {

id

}

}

}

`;

Once more, let’s use the useMutation hook to do the voting. We’ll call the function that runs the mutation vote and will pass the VOTE\_MUTATION GraphQL mutation to it. The vote function will be called in the onClick handler for the div with the up caret (▲) calls upvote button.

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

const Link = (props) => {

*// ...*

const [vote] = **useMutation**(VOTE\_MUTATION, {

variables: {

linkId: link.id

}

});

return (

<div className="flex mt2 items-start">

<div className="flex items-center">

<span className="gray">{props.index + 1}.</span>

<div

className="ml1 gray f11"

style={{ cursor: 'pointer' }}

onClick={vote}

>

▲

</div>

</div>

<div className="ml1">

<div>

{link.description} ({link.url})

</div>

{authToken && (

<div className="f6 lh-copy gray">

{link.votes.length} votes | by{' '}

{link.postedBy ? link.postedBy.name : 'Unknown'}{' '}

{**timeDifferenceForDate**(link.createdAt)}

</div>

)}

</div>

</div>

);

};

This step should feel pretty familiar by now. The onClick handler of the div with the up caret calls the vote function which runs the mutation to place a vote.

We need to import useMutation and gql for the mutation to work.

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

import { useMutation, gql } from '@apollo/client';

We can now go and test the implementation! Run yarn start in hackernews-react-apollo and click the upvote button on a link. You’re not getting any UI feedback yet, but after refreshing the page we’ll see the added votes.

**Remember**: We have to be logged in to being able to vote links!

In the next section, we’ll learn how to automatically update the UI after each mutation!

## Updating the cache

One of Apollo’s biggest value propositions is that it creates and maintains a client-side cache for our GraphQL apps. We typically don’t need to do much to manage the cache, but in some circumstances, we do.

When we perform mutations that affect a list of data, we need to manually intervene to update the cache. We’ll implement this functionality by using the [update callback](https://www.apollographql.com/docs/react/data/mutations/#the-update-function) of useMutation.

Open Link.js and update the mutation to include some additional behavior in the update callback. This runs after the mutation has completed and allows us to read the cache, modify it, and commit the changes.

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

const Link = (props) => {

*// ...*

const [vote] = **useMutation**(VOTE\_MUTATION, {

variables: {

linkId: link.id

},

update: (cache, {data: {vote}}) => {

const { feed } = cache.**readQuery**({

query: FEED\_QUERY

});

const updatedLinks = feed.links.**map**((feedLink) => {

if (feedLink.id === link.id) {

return {

...feedLink,

votes: [...feedLink.votes, vote]

};

}

return feedLink;

});

cache.**writeQuery**({

query: FEED\_QUERY,

data: {

feed: {

links: updatedLinks

}

}

});

}

});

*// ...*

};

In the update callback is that we’ve included with the mutation, we’re calling cache.readQuery and passing in the FEED\_QUERY document. This allows us to read the exact portion of the Apollo cache that we need to allow us to update it. Once we have the cache, we create a new array of data that includes the vote that was just made. The vote that was made with the mutation is [destructured](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) out using { data: { vote } }. Once we have the new list of votes, we can commit the changes to the cache using cache.writeQuery, passing in the new data.

The last thing we need to do for this to work is import the FEED\_QUERY into the Link file:

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

import { FEED\_QUERY } from './LinkList';

That’s it! The update function will now be executed and make sure that the store gets updated properly after a mutation was performed. The store update will trigger a rerender of the component and thus update the UI with the correct information!

While we’re at it, let’s also implement update for adding new links!

Open CreateLink.js and following what we did before, add an update callback to the useMutation hook to update the Apollo store.

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

const [createLink] = **useMutation**(CREATE\_LINK\_MUTATION, {

variables: {

description: formState.description,

url: formState.url

},

update: (cache, { data: { post } }) => {

const data = cache.**readQuery**({

query: FEED\_QUERY,

});

cache.**writeQuery**({

query: FEED\_QUERY,

data: {

feed: {

links: [post, ...data.feed.links]

}

},

});

},

onCompleted: () => **navigate**("/")

});

The update function works in a very similar way as before. We first read the current state of the results of the FEED\_QUERY. Then we insert the newest link at beginning and write the query results back to the store. Note that we need to pass in a set of variables to the readQuery and writeQuery functions. It’s not enough to simply pass the FEED\_QUERY query document in, we also need to specify the conditions of the original query we’re targeting. In this case, we pass in variables that line up with the initial variables we passed into the query in LinkList.js.

We need to import the FEED\_QUERY into the file.

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

import { FEED\_QUERY } from './LinkList';

Awesome, now the store also updates with the right information after new links are added. The app is getting into shape 🤓

# React + Apollo – Filtering: Searching the List of Links

<https://www.howtographql.com/react-apollo/7-filtering-searching-the-list-of-links/>

In this section, we’ll implement a search feature and learn about the filtering capabilities of our GraphQL API.

## Preparing the React components

The search will be available under a new route and implemented in a new React component.

Start by creating a new file called Search.js in src/components and add the following code:

[**.../hackernews-react-apollo/src/components/Search.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Search.js)

import React, { useState } from 'react';

import { useLazyQuery, gql } from '@apollo/client';

import Link from './Link';

const Search = () => {

const [searchFilter, setSearchFilter] = **useState**('');

return (

<>

<div>

Search

<input

type="text"

onChange={(e) => **setSearchFilter**(e.target.value)}

/>

<button>OK</button>

</div>

{data &&

data.feed.links.**map**((link, index) => (

<Link key={link.id} link={link} index={index} />

))}

</>

);

};

export default Search;

Again, this is a pretty standard setup. You’re rendering an input field where the user can type a search string.

The Search component uses the useState hook to hold a search term supplied by the user. The setSearchFilter functions is called in the onChange event on the input to update this value.

The component is also looking for a variable called data which it iterates over to render Link components with the search results. We’ll define and execute the query a bit later.

Let’s now add the Search component as a new route to the app. Open App.js and update it to look as follows:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

const App = () => (

<div className="center w85">

<Header />

<div className="ph3 pv1 background-gray">

<Routes>

<Route

path="/"

element={<Navigate replace to="/new/1" />}

/>

<Route

path="/create"

element={<CreateLink/>}

/>

<Route path="/login" element={<Login/>}/>

<Route path="/search"element={<Search/>}/>

</Routes>

</div>

</div>

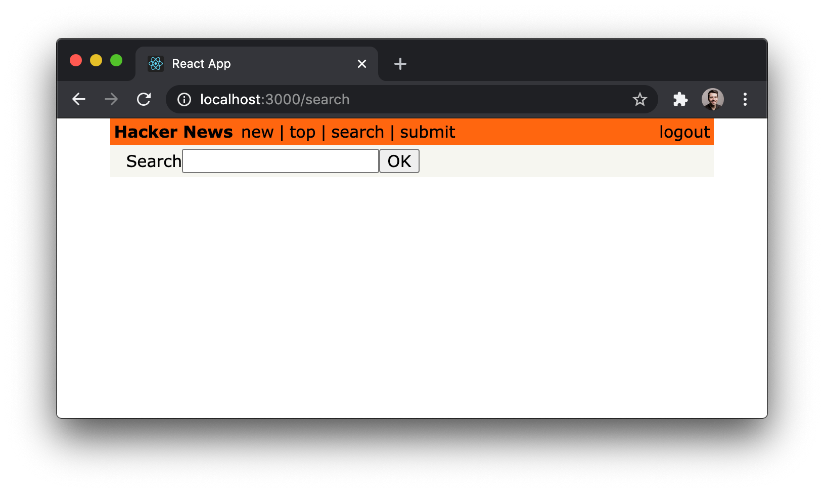
);

Also import the Search component at the top of the file:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

import Search from './Search';

We can now navigate to the search feature using the search button in the Header:



Great, let’s now go back to the Search component and see how we can implement the actual search.

## Filtering Links

Open Search.js and add the following query definition at the top of the file:

[**.../hackernews-react-apollo/src/components/Search.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Search.js)

const FEED\_SEARCH\_QUERY = gql`

query FeedSearchQuery($filter: String!) {

feed(filter: $filter) {

id

links {

id

url

description

createdAt

postedBy {

id

name

}

votes {

id

user {

id

}

}

}

}

}

`;

This query looks similar to the feed query that’s used in LinkList. However, this time it takes in an argument called filter that will be used to constrain the list of links we want to retrieve.

The actual filter is built and used in the feed resolver which is implemented in server/src/resolvers/Query.js:

[**.../hackernews-react-apollo/server/src/resolvers/Query.js**](https://github.com/howtographql/react-apollo/blob/master/server/src/resolvers/Query.js)

async function **feed**(parent, args, context, info) {

const where = args.filter

? {

OR: [

{ description: { contains: args.filter } },

{ url: { contains: args.filter } }

]

}

: {};

const links = await context.prisma.link.**findMany**({

where,

skip: args.skip,

take: args.take,

orderBy: args.orderBy

});

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

return {

id: 'main-feed',

links,

count

};

}

module.exports = {

feed

};

**Note**: To understand what’s going on in this resolver, check out the [filtering chapter of the Node tutorial](https://www.howtographql.com/graphql-js/8-filtering-pagination-and-sorting/).

In this case, two where conditions are specified: A link is only returned if either its url contains the provided filter or its description contains the provided filter. Both conditions are combined using Prisma’s OR operator.

Perfect, the query is defined! But this time we actually want to load the data every time the user hits the **OK** button, not upon the initial load of the component. To do this, we’ll use a hook supplied by Apollo called useLazyQuery. This hook performs a query in the same way the useQuery hook does but the difference is that it must be executed manually. This is perfect for our situation––we want to execute the query when the **OK** button is clicked.

Let’s include useLazyQuery and execute it when the **OK** button is clicked.

[**.../hackernews-react-apollo/src/components/Search.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Search.js)

const Search = () => {

const [searchFilter, setSearchFilter] = **useState**('');

const [executeSearch, { data }] = **useLazyQuery**(

FEED\_SEARCH\_QUERY

);

return (

<>

<div>

Search

<input

type="text"

onChange={(e) => **setSearchFilter**(e.target.value)}

/>

<button

onClick={() =>

**executeSearch**({

variables: { filter: searchFilter }

})

}

>

OK

</button>

</div>

{data &&

data.feed.links.**map**((link, index) => (

<Link key={link.id} link={link} index={index} />

))}

</>

);

};

The implementation is almost trivial! We’re executing the FEED\_SEARCH\_QUERY manually and retrieving the links from the response that’s returned by the server. These links are put into the component’s state so that they can be rendered.

Go ahead and test the app by running yarn start in a terminal and navigating to http://localhost:3000/search. Then type a search string into the text field, click the **OK** button and verify the links that are returned fit the filter conditions.

# React + Apollo – Realtime Updates with GraphQL Subscriptions

<https://www.howtographql.com/react-apollo/8-subscriptions/>

This section is all about bringing realtime functionality into the app by using GraphQL subscriptions.

## What are GraphQL Subscriptions?

Subscriptions are a GraphQL feature allowing the server to send data to its clients when a specific event happens. Subscriptions are usually implemented with [WebSockets](https://en.wikipedia.org/wiki/WebSocket), where the server holds a steady connection to the client. This means when working with subscriptions, we’re breaking the Request-Response cycle that is typically used for interactions with the API. Instead, the client now initiates a steady connection with the server by specifying which event it is interested in. Every time this particular event then happens, the server uses the connection to push the expected data to the client.

## Subscriptions with Apollo

When using Apollo, we need to configure our ApolloClient with information about the subscriptions endpoint. This is done by adding another ApolloLink to the Apollo middleware chain. This time, it’s the WebSocketLink from the [@apollo/client/link/ws](https://github.com/apollographql/apollo-link/tree/master/packages/apollo-link-ws) package.

To get started, add subscriptions-transport-ws as a dependency to the app.

Open a terminal and navigate to the project’s root directory. Then execute the following command:

[**$.../hackernews-react-apollo**](https://github.com/howtographql/react-apollo/blob/master/hackernews-react-apollo)

yarn add subscriptions-transport-ws

Next, let’s make sure our ApolloClient instance knows about the subscription server.

Open index.js and add the following imports to the top of the file:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

import { split } from '@apollo/client';

import { WebSocketLink } from '@apollo/client/link/ws';

import { getMainDefinition } from '@apollo/client/utilities';

Let’s now create a new WebSocketLink that represents the WebSocket connection. We’ll use split for proper “routing” of the requests and update the constructor call of ApolloClient like so:

[**.../hackernews-react-apollo/src/index.js**](https://github.com/howtographql/react-apollo/blob/master/src/index.js)

const wsLink = new WebSocketLink({

uri: `ws://localhost:4000/graphql`,

options: {

reconnect: true,

connectionParams: {

authToken: localStorage.**getItem**(AUTH\_TOKEN)

}

}

});

const link = **split**(

({ query }) => {

const { kind, operation } = **getMainDefinition**(query);

return (

kind === 'OperationDefinition' &&

operation === 'subscription'

);

},

wsLink,

authLink.**concat**(httpLink)

);

const client = new ApolloClient({

link,

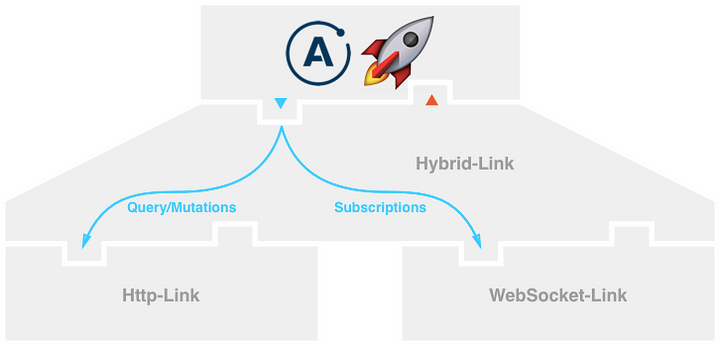
cache: new InMemoryCache()

});

We’re instantiating a WebSocketLink that knows about the subscriptions endpoint. The subscriptions endpoint in this case is similar to the HTTP endpoint, except that it uses the ws (WebSocket) protocol instead of http. Notice that we’re also authenticating the WebSocket connection with the user’s token that we retrieve from localStorage.

[split](https://github.com/apollographql/apollo-link/blob/98eeb1deb0363384f291822b6c18cdc2c97e5bdb/packages/apollo-link/src/link.ts#L33) is used to “route” a request to a specific middleware link. It takes three arguments, the first one is a test function which returns a boolean. The remaining two arguments are again of type ApolloLink. If test returns true, the request will be forwarded to the link passed as the second argument. If false, to the third one.

In our case, the test function is checking whether the requested operation is a subscription. If it is, it will be forwarded to the wsLink, otherwise (if it’s a query or mutation), the authLink.concat(httpLink) will take care of it:



Picture taken from [Apollo Link: The modular GraphQL network stack](https://dev-blog.apollodata.com/apollo-link-the-modular-graphql-network-stack-3b6d5fcf9244) by [Evans Hauser](https://twitter.com/EvansHauser)

## Subscribing to New links

For the app to update in realtime when new links are created, we need to subscribe to events that are happening on the Link type. We’ll implement the subscription in the LinkList component since that’s where all the links are rendered.

The useQuery hook provided by Apollo gives us access to a function called subscribeToMore. We can destructure this function out and use it to act on new data that comes in over a subscription. This will give us the effect of making our app “realtime”.

Open LinkList.js and update current component as follow:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const LinkList = () => {

const {

data,

loading,

error,

subscribeToMore

} = **useQuery**(FEED\_QUERY);

**subscribeToMore**({

*// ...*

});

*// ...*

};

The subscribeToMore function takes a single object as an argument. This object requires configuration for how to listen for and respond to a subscription.

At the very least, we need to pass a subscription document to the document key in this object. This is a GraphQL document where we define our subscription.

We can also pass a field called updateQuery which can be used to update the cache, much like we would do in a mutation.

Let’s get started by providing the complete configuration we need for subscribeToMore to function properly.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

*// ...*

**subscribeToMore**({

document: NEW\_LINKS\_SUBSCRIPTION,

updateQuery: (prev, { subscriptionData }) => {

if (!subscriptionData.data) return prev;

const newLink = subscriptionData.data.newLink;

const exists = prev.feed.links.**find**(

({ id }) => id === newLink.id

);

if (exists) return prev;

return Object.**assign**({}, prev, {

feed: {

links: [newLink, ...prev.feed.links],

count: prev.feed.links.length + 1,

\_\_typename: prev.feed.\_\_typename

}

});

}

});

The definition of updateQuery is somewhat similar to that of update defined in Login.js and CreateLink.js.

The last thing we need to do for this to work is add the NEW\_LINKS\_SUBSCRIPTION to the top of the file:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const NEW\_LINKS\_SUBSCRIPTION = gql`

subscription {

newLink {

id

url

description

createdAt

postedBy {

id

name

}

votes {

id

user {

id

}

}

}

}

`;

The NEW\_LINKS\_SUBSCRIPTION will use the subscription operation of the GraphQL server to listen for any newly created links.

Now we can test our implementation by opening two browser windows. In the first window, we have our application running on http://localhost:3000/. In the second window, we can open the GraphQL Playground running in http://localhost:4000/ and send a post mutation. Here is an example mutation you can try:

mutation {

post(url: "www.graphqlweekly.com", description: "A weekly newsletter about GraphQL") {

id

}

}

When you send the mutation, you should see the app update in realtime! ⚡️

## Subscribing to New Votes

We can also subscribe to new votes that are submitted by other users so that the latest vote count is always visible in the app.

Open LinkList.js and add another subscribeToMore method call to the LinkList component:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const LinkList = () => {

*// ...*

**subscribeToMore**({

document: NEW\_VOTES\_SUBSCRIPTION

});

*// ...*

Similar to what we did before, we’re calling subscribeToMore but now using NEW\_VOTES\_SUBSCRIPTION as the document. This time, we’re passing in a subscription that asks for newly created votes. When the subscription fires, Apollo Client automatically updates the link that was voted on.

Still in LinkList.js add the NEW\_VOTES\_SUBSCRIPTION to the top of the file:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const NEW\_VOTES\_SUBSCRIPTION = gql`

subscription {

newVote {

id

link {

id

url

description

createdAt

postedBy {

id

name

}

votes {

id

user {

id

}

}

}

user {

id

}

}

}

`;

Fantastic! Our app is now ready for realtime and will immediately update links and votes whenever they’re created by other users.

**Note:** If you want to learn more about how subscriptions are implemented in the GraphQL server, check out the [subscriptions chapter of the Node tutorial](https://www.howtographql.com/graphql-js/7-subscriptions/).

# React + Apollo – Pagination

<https://www.howtographql.com/react-apollo/9-pagination/>

The last topic that we’ll cover in this tutorial is pagination. We’ll implement a simple pagination approach so that users are able to view the links in smaller chunks rather than having an extremely long list of Link elements.

## Preparing the React Components

Once more, we first need to prepare the React components for this new functionality. In fact, we’ll make a slight adjustment to the current routing setup. The idea is that the LinkList component will be used for two different purposes (and routes). The first one is to display the top ten voted links and the second use case is to display new links in a list separated into multiple pages that the user can navigate through.

Open App.js and adjust the component to look like this:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

const App = () => (

<div className="center w85">

<Header />

<div className="ph3 pv1 background-gray">

<Routes>

<Route

path="/"

element={<Navigate replace to="/new/1" />}

/>

<Route

path="/create"

element={<CreateLink/>}

/>

<Route path="/login" element={<Login/>}/>

<Route path="/search"element={<Search/>}/>

<Route path="/top" element={<LinkList/>} />

<Route

path="/new/:page"

element={<LinkList/>}

/>

</Routes>

</div>

</div>

);

Let’s be sure to import the Navigate component so we don’t get any errors.

Update the router import on the top of the file:

[**.../hackernews-react-apollo/src/components/App.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/App.js)

import {Navigate, Route, Routes} from 'react-router-dom';

We’ve now added two new routes: /top and /new/:page. The latter reads the value for page from the url so that this information is available inside the component that’s rendered. For this route that’s LinkList.

The main route / now redirects to the first page of the route where new posts are displayed.

Before moving on, quickly add a new navigation item to the Header component that brings the user to the /top route.

Open Header.js and add the following lines between the / and the /search routes:

[**.../hackernews-react-apollo/src/components/Header.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Header.js)

<Link to="/" className="ml1 no-underline black">

new

</Link>

<div className="ml1">|</div>

<Link to="/top" className="ml1 no-underline black">

top

</Link>

<div className="ml1">|</div>

<Link

to="/search"

className="ml1 no-underline black"

>

We also need to add some logic to the LinkList component to account for the two different responsibilities it now has.

Open LinkList.js and add three arguments to the FeedQuery by replacing the FEED\_QUERY definition with the following:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

export const FEED\_QUERY = gql`

query FeedQuery(

$take: Int

$skip: Int

$orderBy: LinkOrderByInput

) {

feed(take: $take, skip: $skip, orderBy: $orderBy) {

id

links {

id

createdAt

url

description

postedBy {

id

name

}

votes {

id

user {

id

}

}

}

count

}

}

`;

The query now accepts arguments that we’ll use to implement pagination and ordering. skip defines the offset where the query will start. For example, if we passed a value of **10** for this argument, it means that the first 10 items of the list will not be included in the response. take then defines the limit or how many elements we want to load from that list. If we pass in 10 for skip and 5 for take, we’ll receive items 10 to 15 from the list. orderBy defines how the returned list should be sorted.

But how can we pass the variables when using the useQuery hook which is fetching the data under the hood? The key is that we need to pass these variables in when we make the call to useQuery.

Still in LinkList.js, adjust the useQuery hook to accept the variables we want to pass to the query.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

import { useLocation } from 'react-router-dom';

// ...

const LinkList = () => {

const location = useLocation();

const isNewPage = location.pathname.includes(

'new'

);

const pageIndexParams = location.pathname.split(

'/'

);

const page = parseInt(

pageIndexParams[pageIndexParams.length - 1]

);

const pageIndex = page ? (page - 1) \* LINKS\_PER\_PAGE : 0;

const {

data,

loading,

error,

subscribeToMore

} = useQuery(FEED\_QUERY, {

variables: getQueryVariables(isNewPage, page),

});

// ...

};

We use the useLocation hook to get the current pathname of the page being visited.

We’re passing in an object as the second argument to useQuery, right after we pass in the FEED\_QUERY document. We can use this object to modify the behavior of the query in various ways. One of the most common things we do with it is to provide variables.

The variables key points to a function call that will retrieve the variables. Let’s implement the getQueryVariables function.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const getQueryVariables = (isNewPage, page) => {

const skip = isNewPage ? (page - 1) \* LINKS\_PER\_PAGE : 0;

const take = isNewPage ? LINKS\_PER\_PAGE : 100;

const orderBy = { createdAt: 'desc' };

return { take, skip, orderBy };

};

The getQueryVariables function is responsible for returning values for skip, take, and orderBy. For skip, we first check whether we are currently on the /new route. If so, the value for skip is the current page (subtracting 1 to handle the index) multiplied by the LINKS\_PER\_PAGE contstant. If we’re not on the /new route, the value for skip is 0. We use the same LINKS\_PER\_PAGE constant to determine how many links to take.

We’re now passing take, skip, orderBy values as variables based on the current page.

Also note that we’re including the ordering attribute { createdAt: 'desc' } for the new page to make sure the newest links are displayed first. The ordering for the /top route will be calculated manually based on the number of votes for each link.

We also need to define the LINKS\_PER\_PAGE constant and then import it into the LinkList component.

Open src/constants.js and add the following definition:

[**.../hackernews-react-apollo/src/constants.js**](https://github.com/howtographql/react-apollo/blob/master/src/constants.js)

export const LINKS\_PER\_PAGE = 5;

Now adjust the import statement from ../constants in LinkList.js to also include the new constant:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

import { LINKS\_PER\_PAGE } from '../constants';

### Implementing Navigation

Next, we need functionality for the user to switch between the pages. First add two button elements to the bottom of the LinkList component that can be used to navigate back and forth.

Open LinkList.js and update the returned JSX to look as follows:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

return (

<>

{loading && <p>Loading...</p>}

{error && <pre>{JSON.stringify(error, null, 2)}</pre>}

{data && (

<>

{getLinksToRender(isNewPage, data).map(

(link, index) => (

<Link

key={link.id}

link={link}

index={index + pageIndex}

/>

)

)}

{isNewPage && (

<div className="flex ml4 mv3 gray">

<div

className="pointer mr2"

onClick={() => {

if (page > 1) {

navigate(`/new/${page - 1}`);

}

}}

>

Previous

</div>

<div

className="pointer"

onClick={() => {

if (

page <=

data.feed.count / LINKS\_PER\_PAGE

) {

const nextPage = page + 1;

navigate(`/new/${nextPage}`);

}

}}

>

Next

</div>

</div>

)}

</>

)}

</>

);

Since the setup is slightly more complicated now, we are going to calculate the list of links to be rendered in a separate method.

Still in LinkList.js, add the following method implementation:

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

const getLinksToRender = (isNewPage, data) => {

if (isNewPage) {

return data.feed.links;

}

const rankedLinks = data.feed.links.slice();

rankedLinks.sort(

(l1, l2) => l2.votes.length - l1.votes.length

);

return rankedLinks;

};

For the /new route, we simply return all the links returned by the query. That’s logical since here we don’t have to make any manual modifications to the list that is to be rendered. If the user loaded the component from the /top route, we’ll sort the list according to the number of votes and return the top 10 links.

Finally, we need to add the useNavigate hook to LinkList

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

import {useLocation, useNavigate} from 'react-router-dom';

// ...

const LinkList = () => {

const navigate = useNavigate();

// ...

}

Let’s have a closer look at the logic for the **Next** and **Previous** links.

[**.../hackernews-react-apollo/src/components/LinkList.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/LinkList.js)

{

isNewPage && (

<div className="flex ml4 mv3 gray">

<div

className="pointer mr2"

onClick={() => {

if (page > 1) {

navigate(`/new/${page - 1}`);

}

}}

>

Previous

</div>

<div

className="pointer"

onClick={() => {

if (page <= data.feed.count / LINKS\_PER\_PAGE) {

const nextPage = page + 1;

navigate(`/new/${nextPage}`);

}

}}

>

Next

</div>

</div>

);

}

We start by retrieving the current page from the URL and doing a sanity check to make sure that it makes sense to paginate back or forth. We then calculate the next page and tell the router where to navigate to next. The router will then reload the component with a new page in the URL that will be used to calculate the right chunk of links to load.

Run the app by typing yarn start in a terminal and use the new buttons to paginate through the list of links!

### Final Adjustments

Through the changes that we made to the FEED\_QUERY, we’ll notice that the update functions of the mutations don’t work any more. That’s because readQuery now also expects to get passed the same variables that we defined before.

**Note**: readQuery essentially works in the same way as the query method on the ApolloClient that we used to implement the search. However, instead of making a call to the server, it will simply resolve the query against the local store! If a query was fetched from the server with variables, readQuery also needs to know the variables to make sure it can deliver the right information from the cache.

With that information, open Link.js and update the update function on the useMutation hook:

[**.../hackernews-react-apollo/src/components/Link.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/Link.js)

import { AUTH\_TOKEN, LINKS\_PER\_PAGE } from '../constants';

// ...

const take = LINKS\_PER\_PAGE;

const skip = 0;

const orderBy = { createdAt: 'desc' };

const [vote] = useMutation(VOTE\_MUTATION, {

variables: {

linkId: link.id

},

update: (cache, {data: {vote}}) => {

const { feed } = cache.readQuery({

query: FEED\_QUERY,

variables: {

take,

skip,

orderBy

}

});

const updatedLinks = feed.links.map((feedLink) => {

if (feedLink.id === link.id) {

return {

...feedLink,

votes: [...feedLink.votes, vote]

};

}

return feedLink;

});

cache.writeQuery({

query: FEED\_QUERY,

data: {

feed: {

links: updatedLinks

}

},

variables: {

take,

skip,

orderBy

}

});

}

});

Similarly update the update function of the useMutation hook in CreateLink.js

[**.../hackernews-react-apollo/src/components/CreateLink.js**](https://github.com/howtographql/react-apollo/blob/master/src/components/CreateLink.js)

import { AUTH\_TOKEN, LINKS\_PER\_PAGE } from '../constants';

// ...

const [createLink] = useMutation(CREATE\_LINK\_MUTATION, {

variables: {

description: formState.description,

url: formState.url

},

update: (cache, {data: {post}}) => {

const take = LINKS\_PER\_PAGE;

const skip = 0;

const orderBy = {createdAt: 'desc'};

const data = cache.readQuery({

query: FEED\_QUERY,

variables: {

take,

skip,

orderBy

}

});

cache.writeQuery({

query: FEED\_QUERY,

data: {

feed: {

links: [post, ...data.feed.links]

}

},

variables: {

take,

skip,

orderBy

}

});

},

onCompleted: () => {

navigate("/")

}

});

We have now added a simple pagination system to the app, allowing users to load links in small chunks instead of loading them all up front.

# GraphQL.NET

## Introduction

<https://graphql-dotnet.github.io/docs/getting-started/introduction/>

**Samples**:

* <https://www.codemag.com/Article/2307051/Building-Microservices-Architecture-Using-GraphQL-and-ASP.NET-7-Core>
* <https://www.codemag.com/Article/1909061/Intro-to-GraphQL-for-.NET-Developers-Schema-Resolver-and-Query-Language> (*outdated*)
* <https://www.codemag.com/Article/2003051/Introduction-to-GraphQL-for-.NET-Developers-Mutation> (outdated)
* <https://chillicream.com/docs/hotchocolate/v13/get-started-with-graphql-in-net-core>
* <https://github.com/ChilliCream/graphql-workshop/blob/master/docs/1-creating-a-graphql-server-project.md>
* <https://dotnetthoughts.net/getting-started-with-graphql-aspnetcore/>
* <https://code-maze.com/graphql-mutations/>

**Solution:** [https://github.com/AjaySingala/dotNetFullStackDemos/tree/main/GraphQL.NET/GraphQL-Intro/GeneralDemos.sln](https://github.com/AjaySingala/dotNetFullStackDemos/tree/main/GeneralDemos/GeneralDemos.sln)

## Installation

<https://graphql-dotnet.github.io/docs/getting-started/installation/>

## GraphiQL – IDE

<https://graphql-dotnet.github.io/docs/getting-started/graphiql/>

## Altair GraphQL Client

<https://graphql-dotnet.github.io/docs/getting-started/altair-graphql/>

## Queries

<https://graphql-dotnet.github.io/docs/getting-started/queries/>

## Schema Types

<https://graphql-dotnet.github.io/docs/getting-started/schema-types/>

## Custom Scalars

<https://graphql-dotnet.github.io/docs/getting-started/custom-scalars/>

## Lists and Non-Null

<https://graphql-dotnet.github.io/docs/getting-started/lists-non-null/>

## Arguments

<https://graphql-dotnet.github.io/docs/getting-started/arguments/>

## Aliases

<https://graphql-dotnet.github.io/docs/getting-started/arguments/>

## Fragments

<https://graphql-dotnet.github.io/docs/getting-started/fragments/>

## Variables

<https://graphql-dotnet.github.io/docs/getting-started/variables/>

## Directives

<https://graphql-dotnet.github.io/docs/getting-started/directives/>

## Mutations

<https://graphql-dotnet.github.io/docs/getting-started/mutations/>

## Interfaces

<https://graphql-dotnet.github.io/docs/getting-started/interfaces/>

## Unions

<https://graphql-dotnet.github.io/docs/getting-started/unions/>

## Subscriptions

<https://graphql-dotnet.github.io/docs/getting-started/subscriptions/>

## Query Validation

<https://graphql-dotnet.github.io/docs/getting-started/query-validation/>

## Query Organization

<https://graphql-dotnet.github.io/docs/getting-started/query-organization/>

## User Context

<https://graphql-dotnet.github.io/docs/getting-started/user-context/>

## Error Handling

<https://graphql-dotnet.github.io/docs/getting-started/errors/>

## Dependency Injection

https://graphql-dotnet.github.io/docs/getting-started/dependency-injection/

## How do I use XYZ ORM/database with GraphQL.NET?

<https://graphql-dotnet.github.io/docs/getting-started/databases/>

## Protection Against Malicious Queries

<https://graphql-dotnet.github.io/docs/getting-started/malicious-queries/>

## Object/Field Metadata

<https://graphql-dotnet.github.io/docs/getting-started/metadata/>

## Field Middleware

<https://graphql-dotnet.github.io/docs/getting-started/field-middleware/>

## Metrics

https://graphql-dotnet.github.io/docs/getting-started/metrics/

## Authorization

<https://graphql-dotnet.github.io/docs/getting-started/authorization/>

## Global Switches

<https://graphql-dotnet.github.io/docs/getting-started/global-switches/>

# Intro to GraphQL for .NET Developers: Schema, Resolver, and Query Language

<https://www.codemag.com/Article/1909061/Intro-to-GraphQL-for-.NET-Developers-Schema-Resolver-and-Query-Language>

# Implementing a TechEvent Management GraphQL with ASP.Net Core

<https://www.c-sharpcorner.com/article/building-api-in-net-core-with-graphql2/>

# .Net 5 API with GraphQL - Step by Step Using Hot Chocolate

<https://dev.to/moe23/net-5-api-with-graphql-step-by-step-2b20>

# How to implement GraphQL in ASP.Net Core Web API (.NET 6) using HotChocolate

<https://blog.christian-schou.dk/how-to-implement-graphql-in-asp-net-core/>