

# Advanced GraphQL with Apollo

Build a Distributed GraphQL API with  
Apollo Federation 2 and Apollo Server

Second Edition

Mandi Wise

# **Advanced GraphQL with Apollo**

**Build a Distributed GraphQL API with Apollo Federation 2 and Apollo Server**

**Mandi Wise**



**8-bit press**

## **Advanced GraphQL with Apollo**

By Mandi Wise

Copyright © 2022 8-Bit Press Inc. All rights reserved.

Published by 8-Bit Press Inc.

<https://8bit.press>

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form by means electronic, mechanical, photocopying, or otherwise without out prior written permission of the publisher, except for brief quotations embodied in articles or reviews. Thank you for respecting the hard work of the author.

While the advice and information in this book is believed to be true and accurate at the date of publication, the publisher and the author assume no legal responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

First Edition, April 2020

Second Edition, June 2022

**Illustrator:** Mandi Wise

Cover by Mandi Wise

ISBN 978-1-7782366-0-0

*For James*



# Contents

<b>Preface</b>	<b>ix</b>
What's Inside . . . . .	ix
Who Should Read this Book . . . . .	x
Getting the Most from this Book . . . . .	x
Formatting Conventions . . . . .	xi
Code Blocks . . . . .	xi
Inline Code . . . . .	xi
Info Boxes . . . . .	xii
Quotes . . . . .	xii
Package Versions . . . . .	xii
Reference Source Code. . . . .	xii
Required Software . . . . .	xii
Node.js . . . . .	xii
MongoDB . . . . .	xiii
Docker . . . . .	xiii
Rover CLI. . . . .	xiii
Optional Software . . . . .	xiv
The Game Plan . . . . .	xiv
 <b>Chapter 1   Apollo Federation and Gateway</b>	 <b>1</b>
A Basic Apollo Server. . . . .	1
Organize the Server Files . . . . .	5
Why Use Apollo Federation? . . . . .	7
But First, More Refactoring. . . . .	9
Compose a Subgraph into the Gateway . . . . .	13
The First Entity . . . . .	18
Optional: Supergraph Composition with the Rover CLI . . . . .	21
Summary. . . . .	23

<b>Chapter 2   Authentication and User Account Management with Auth0</b>	<b>24</b>
Sign Up for Auth0 . . . . .	24
What Are JSON Web Tokens and How Do They Work? . . . . .	27
Check JWTs with Express Middleware . . . . .	28
Pass Context from the Gateway and the Accounts Service . . . . .	32
Create Auth0 Applications, APIs, and Users . . . . .	35
Generate a Token for Explorer . . . . .	41
The Auth0 Management API . . . . .	46
Configure Auth0 with Node.js . . . . .	49
Add New Account Fields . . . . .	51
Add a <code>createAccount</code> Mutation . . . . .	56
Add Mutations to Update Accounts . . . . .	58
Add a <code>deleteAccount</code> Mutation . . . . .	62
Summary . . . . .	64
<b>Chapter 3   Apollo Data Sources, Custom Scalars, and Custom Directives</b>	<b>65</b>
Manage Data-Fetching Logic in an Apollo Data Source . . . . .	65
Add a Custom <code>DateTime</code> Scalar Type . . . . .	72
Approaches to Authorization in GraphQL . . . . .	78
Custom Directives with Apollo Federation . . . . .	79
Handling Authorization with Type System Directives . . . . .	80
Securing References Resolvers . . . . .	88
Summary . . . . .	90
<b>Chapter 4   User Metadata Management with MongoDB and Mongoose</b>	<b>92</b>
Why Create a Separate Service for User Metadata? . . . . .	92
Install Packages and Configure Mongoose . . . . .	93
Scaffold the Profiles Service . . . . .	97
Add the <code>Profile</code> Type and Its Query Fields . . . . .	101
Add a <code>createProfile</code> Mutation . . . . .	108
Add <code>updateProfile</code> and <code>deleteProfile</code> Mutations . . . . .	116
Add and Remove Users from Another User's Network . . . . .	121
Add a Search Query with Full-Text Search in MongoDB . . . . .	130
Summary . . . . .	135
<b>Chapter 5   Relay-Style Pagination</b>	<b>136</b>
Pagination Primer . . . . .	136
Offset-Based . . . . .	136
Cursor-Based . . . . .	138
Relay-Style . . . . .	139
Relay-Style Pagination Internals . . . . .	140
Forward Pagination with Ascending Sort Order . . . . .	141
Forward Pagination with Descending Sort Order . . . . .	143

Backward Pagination with Ascending Sort Order . . . . .	144
Backward Pagination with Descending Sort Order . . . . .	146
Add Pagination Types to the Profiles Service . . . . .	147
Create a Pagination Class for MongoDB Data . . . . .	152
Add Forward Pagination to the network Field . . . . .	162
Add Backward Pagination to the network Field . . . . .	171
Generate the PageInfo to Include in the Response . . . . .	177
Add Pagination to profiles and searchProfiles Fields . . . . .	182
Summary . . . . .	197
 <b>Chapter 6   Bookmark Management with MongoDB and Mongoose</b>	 <b>199</b>
Install Packages and Configure Mongoose . . . . .	199
Scaffold the Bookmarks Service . . . . .	202
Add the Bookmark Type and Its Query Fields . . . . .	206
Add a <code>createBookmark</code> Mutation . . . . .	219
Add <code>updateBookmark</code> and <code>deleteBookmark</code> Mutations . . . . .	228
Get Recommended Bookmarks Based on a User's Interests . . . . .	235
Summary . . . . .	240
 <b>Chapter 7   API Performance and Security Considerations</b>	 <b>241</b>
Potential Performance and Security Issues . . . . .	241
Configure Automatic Persisted Queries . . . . .	242
Limit Query Depth . . . . .	247
Batch Database Queries Using DataLoader . . . . .	250
Restrict API Discoverability . . . . .	258
Summary . . . . .	261
 <b>Chapter 8   Multi-Subgraph Workflows with Temporal</b>	 <b>262</b>
The Challenge of Cascading User Data Deletion Across Services . . . . .	262
Install Packages and Scaffold the Workflows Service . . . . .	265
Create Another Auth0 Application to Authenticate the Workflows Service . . . . .	269
Add a New Authorization Directive to Handle Token Scope . . . . .	277
Define a Multi-Subgraph Workflow with Temporal . . . . .	283
Add a <code>deleteAllUserData</code> Mutation Field . . . . .	292
Summary . . . . .	297
 <b>Chapter 9   Managed Federation with Apollo Studio</b>	 <b>298</b>
Why Use Managed Federation? . . . . .	298
Publish Subgraphs to a Deployed Graph in Apollo Studio . . . . .	299
Configure Apollo Gateway to Run in Managed Mode . . . . .	305
Safely Evolve Subgraphs with Schema Checks . . . . .	311
Summary . . . . .	315

<b>Chapter 10   Apollo Router</b>	<b>316</b>
Configure Apollo Router. . . . .	316
Optional: Additional Helpful Docker Commands. . . . .	321
Set Up a Proxy to Handle Authentication . . . . .	322
Summary. . . . .	328
<b>About the Author</b>	<b>329</b>
<b>Changelog</b>	<b>330</b>

# Preface

Welcome to the second edition of *Advanced GraphQL with Apollo!* I'm very excited that you've chosen this book to broaden your GraphQL skill set.

GraphQL was invented at Facebook in 2012 and was publicly launched in 2015. Since then, it has opened up a world of new possibilities for API design and consumption. Today, it has been famously adopted by companies including The New York Times, Shopify, Coursera, AirBnB, GitHub, and Netflix. It has also inspired countless open source projects related to it, including the popular suite of Apollo tools for working with GraphQL APIs in both client and server applications.

And as adoption has expanded so have the demands on what is possible with a GraphQL API, especially when it's used to support distributed application architectures. GraphQL shines in how it allows you to consolidate data from multiple sources in field resolver functions to support specific client use cases. But over time, and as more and more teams begin to contribute to a monolithic GraphQL schema, the schema evolution process can become a choke point in everyone's development process. Alternatively, bluntly dividing type definitions between teams and manually reconsolidating those definitions into a single schema for clients to query may be easier said than done because type and field boundaries can be imprecise.

The initial Apollo Federation specification offered a compelling solution to these challenges. It allows a schema to be divided based on separation of concerns and then declaratively composed into a single GraphQL API. The second version of the Apollo Federation specification—made generally available in 2022—augments these features with a more flexible composition model to make iterative schema evolution among teams even easier. The chapters that follow will explore many of those features.

## What's Inside

This book covers a wide range of topics relevant to developing distributed GraphQL APIs from scratch with JavaScript and it does so using a practical, project-based approach. Working through this book from start to finish will leave you with a relatively full-featured GraphQL API that resolves data from several different back-end services.

Specific topics covered throughout the book include:

- Composing multiple subgraph schemas into a single GraphQL API using Apollo Gateway
- Adding authentication to a Node.js application with Express middleware and Auth0
- Adding authorization to a GraphQL API using a series of custom type system directives
- Creating custom Scalar types to share across subgraph schemas
- Using Apollo data sources to fetch data in resolvers
- Implementing Relay-style pagination for data from MongoDB
- Securing and optimizing the performance of a GraphQL API
- Using an orchestrator to coordinate mutations across multiple subgraphs
- Using the next-gen Apollo Router as an alternative to Apollo Gateway

In its exploration of these topics, this book is divided into 10 chapters. Chapters 1 to 6 focus on building out three core services that will provide data for the API. Chapters 7-10 focus on more advanced topics including performance and security considerations, mutation orchestration, managed federation, and the Rust-based Apollo Router.

Because of the intermediate nature of the content, this book does not cover the basics of Node.js, or MongoDB and assumes some prior exposure to GraphQL.

## Who Should Read this Book

This book covers more advanced topics than a typical introductory GraphQL book or course, but the topics are presented in an accessible way by building up code examples piece-by-piece with explanations in the supporting text. That said, before proceeding, consider if you have at least:

- Intermediate knowledge of JavaScript (including ES2015+ features of the language)
- Previous exposure to GraphQL, such as basic schema design concepts, using a GraphQL API to read and write some data before, and creating a resolver function in Apollo Server
- Experimented with Node.js and MongoDB in the context of web application development
- Some awareness of the advantages of GraphQL APIs compared to other options (there's no preamble GraphQL sales pitch in this book—we're going to jump right into code)

If those qualifications sound like they describe you, then you're in the right place. By the time you have finished this book, you'll feel confident using advanced features of Apollo Server and Apollo Federation and applying what you learn here to complex, real-world development scenarios.

## Getting the Most from this Book

To get the most from your experience reading this book I encourage you to work through each chapter chronologically because the code in each chapter builds on the previous one. This book is structured sequentially around building out a distributed GraphQL API from scratch.

If you intend to build out the demonstration application as you work through the chapters, I also encourage you to explicitly type out as much of the code that you encounter as possible rather

than simply copying and pasting snippets. In my experience, taking the time to intentionally type out code examples aids comprehension and builds mental muscle memory.

Of course, if you’re working on another GraphQL-based project right now and are in search of insights or code examples for any of the aforementioned topics, each chapter of this book also stands alone in building out a coherent chunk of functionality for the application. You will almost certainly find that various chapters serve as helpful reference points for that work too.

## Formatting Conventions

### Code Blocks

In favor of brevity, some code examples are truncated where code that was previously added to a file doesn’t change in the context of a new update to that file. Additionally, file diffs are highlighted in code examples to enhance readability and filenames are added above the code blocks in italics, where applicable. For example:

*gateway/src/index.js*

```
import http from "http";

import app from "./config/app.js";
import initGateway from "./config/apollo.js";

const port = process.env.PORT;
const httpServer = http.createServer(app);
const gateway = initGateway(httpServer);

await gateway.start();
gateway.applyMiddleware({ app, path: "/" });

await new Promise(resolve => httpServer.listen({ port }, resolve));
console.log(`Gateway ready at
  http://localhost:${port}${gateway.graphqlPath}`);
```

### Inline Code

Code and filenames that are referenced inline (for example, within a paragraph or list item) are rendered in this `monospaced` font to distinguish them from the other text.

## Info Boxes

Supplementary notes and additional tips related to the main content are presented in gray boxes throughout the chapters as follows:

You can find a current version of the GraphQL specification here: <http://spec.graphql.org/>

## Quotes

Longer sections of text directly quoted from another source are delineated with a gray border to the left of the quote:

The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct.

—Donald E. Knuth, *Selected Papers on Computer Science*

## Package Versions

For compatibility purposes, package versions are specified wherever you are instructed to install a package from npm. You may wish to experiment with updated package versions as they become available, but compatibility with the other code in this book cannot be guaranteed.

## Reference Source Code

The complete source code for the application built throughout this book may be accessed at:

<https://github.com/8bitpress/advanced-graphql-source-code-v2>

## Required Software

### Node.js

To build the application as outlined in the chapters that follow you will need to have **Node.js 16.x or higher** installed on your computer. You can download the latest LTS version of Node.js for your operating system here:

<https://nodejs.org/en/download/>

## MongoDB

Two of the back-end services implemented in our application will use MongoDB as a database, so you will need to install **MongoDB Community Edition 4.2 or higher** as well. Download the latest version of MongoDB for your operating system here:

<https://docs.mongodb.com/manual/administration/install-community/>

## Docker

### Mac or Windows

Docker is used in Chapters 8 and 10. If you’re using a Mac or Windows computer, then you can download Docker Desktop to install all of the software needed to run Docker here:

<https://docs.docker.com/install/>

You will need to sign up for a free Docker Hub account to download this software. Docker Desktop installs everything you need to run Docker locally, including Docker Engine, the Docker CLI client, Docker Compose, and a dashboard interface for managing Docker containers.

### Linux

If you’re using Linux, then you’ll need to refer to the Docker documentation (using the link above) for details on how to install Docker Engine for your preferred distribution.

You will also need to install Docker Compose for Linux separately afterward:

<https://docs.docker.com/compose/install/>

## Rover CLI

The Rover CLI is used to compose a supergraph schema locally, publish schemas to Apollo Studio for use with managed federation, and check proposed schema changes against known operations.

### Mac or Linux

To install the specific version of Rover used in this book for Mac or Linux, run this command:

```
curl -sSL https://rover.apollo.dev/nix/v0.6.0 | sh
```

### Windows

On Windows machines, run this command to install Rover via the Windows PowerShell Installer:

```
iwr 'https://rover.apollo.dev/win/v0.6.0' | iex
```

### Verify Installation

Once installed for either Mac/Linux or Windows as instructed above, you should be able to run the following command:

```
rover --version
```

And see the following output in the terminal:

```
Rover 0.6.0
```

### Optional Software

To assist with development you may also wish to download and install a GUI for interacting with MongoDB, such as **MongoDB Compass**:

<https://www.mongodb.com/products/compass>

Lastly, if you use VS Code as an editor, then also consider installing the **Apollo GraphQL extension** to enhance your development experience with features such as GraphQL syntax highlighting:

<https://marketplace.visualstudio.com/items?itemName=apollographql.vscode-apollo>

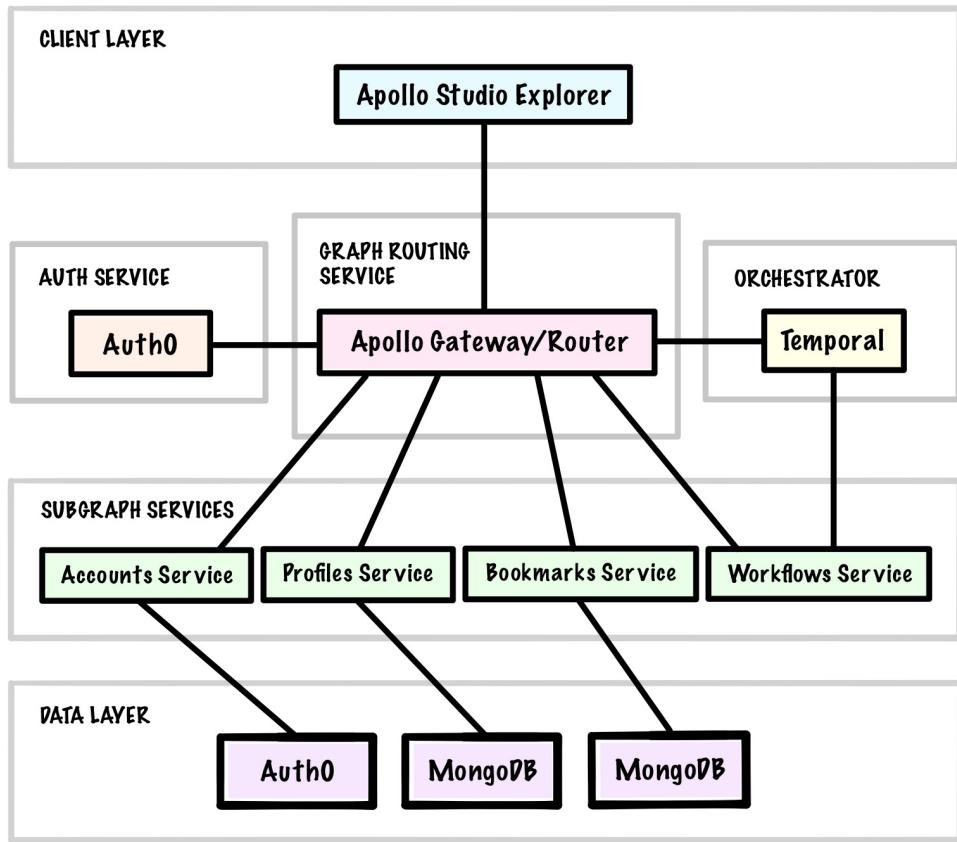
## The Game Plan

Once the required software is installed, you're ready to start coding in Chapter 1. In the pages that follow, we will build out a federated GraphQL API for *Marked*, which is a throwback, Web 2.0-inspired social bookmarking application. Marked users can save public and private bookmarks, add users to their network, and search for other users and public bookmarks using full-text search. As we build Marked, we'll use Apollo Studio Explorer to test out all of the queries and mutations that we add to the schema.

Structurally, Marked will consist of a gateway-level Apollo Server instance that routes incoming GraphQL requests from clients to four different subgraph schemas. Subgraphs can be implemented using a [variety of different GraphQL servers](#) in many different languages, but we will stick with Apollo Server to power all of the subgraph services as well. In the final chapter of the book, we will swap out the Node.js-based Apollo Gateway for the new Rust-based Apollo Router.

Beyond these open-source libraries from Apollo, we will also use Auth0 to provide the authentication layer for the API and store basic user account data, MongoDB will be used as a database for two other services that manage user profile data and bookmarks data, and Temporal will be used to orchestrate cascading deletion of user account data across subgraphs. We will also use Apollo Studio as a schema registry and for viewing operation traces and metrics.

Our project's architecture can be visualized as follows:



The journey of 1,000 miles begins with a single `npm install`, so onward to building our first subgraph schema with Apollo Federation!

## Chapter 1

# Apollo Federation and Gateway

In this chapter, we will:

- Configure a basic development environment
- Set up Express and instantiate a new `ApolloServer`
- Explore Apollo Federation and its use cases
- Create an Apollo Gateway with a single subgraph schema composed into its supergraph schema

## A Basic Apollo Server

One of the main value propositions of Apollo Federation is that it allows you to expose a single GraphQL endpoint to clients while allowing back-end service teams to each manage a logical subsection of the broader schema. From a client developer's perspective, they send their requests to a single GraphQL API, but behind the scenes, we have the flexibility to distribute ownership of the schema's type definitions while composing them together in a single, federated API.

By the end of this book, we'll have an Apollo Server configured for the gateway service plus each of the four subgraph services that we'll eventually build out (refer to the Preface for a detailed architectural overview of the Marked app). To get started, we'll set up a basic Apollo Server, then make some modifications to it to work in conjunction with the `@apollo/gateway` package, and finally create our first subgraph for user accounts data.

We'll begin by creating a top-level project directory to house all of the code for the back-end services we'll build for this project. Inside of that new directory, create a subdirectory called `gateway`. From inside the new `gateway` subdirectory, run the following commands to create a boilerplate `package.json` file and install the initial packages we need to set up Apollo Server:

*gateway/*

```
npm init --yes
npm i apollo-server-express@3.7.0 dotenv@16.0.0 express@4.17.3
  graphql@16.5.0
npm i -D nodemon@2.0.15
```

Eventually, we'll need to use middleware with this server to support user authentication, so instead of installing the `apollo-server` package, we installed `apollo-server-express` and `express`. Doing so will allow us to use Express to integrate Node.js middleware with Apollo Server later on.

We also need the `graphql` peer dependency that's required by Apollo (it's the library used to build a schema and to execute queries against that schema). We also install the `dotenv` package to configure environment variables for this service. Lastly, we install `nodemon` as a development dependency to restart the Node.js process when changes are saved in our project's `gateway` files.

This project will use `import` and `export` syntax to load ES modules so we'll need to declare that in the `package.json` file by setting the `type` property to `module`. We'll also replace the auto-generated scripts with one to start the gateway using `nodemon` with the `dotenv` module preloaded using the `-r` flag as follows:

*gateway/package.json*

```
{
  // ...
  "type": "module",
  "scripts": {
    "dev": "nodemon -r dotenv/config -e env,graphql,js ./src/index.js"
  },
  // ...
}
```

We don't have a `src` subdirectory or an `index.js` file yet, so we'll create `src` inside of `gateway` and then create an `index.js` file inside of that. This `index.js` file will be the main entry point for our gateway code.

Additionally, we'll create a `.env` file at the root of the `gateway` directory. With the help of the `dotenv` package we just installed, the `.env` file will allow us to define environment-specific variables for the Marked application. These values will be accessible on the `process.env` global variable, which is loaded into the Node.js process at runtime.

Note that `.env` files often contain sensitive values such as API keys, so if you're using Git for version control in this project, then be sure to adhere to the best practice of adding the `.env` file to your `.gitignore` file before your next commit.

Our first two variables will be `NODE_ENV` to identify our environment as `development` and `PORT` to identify the port where our GraphQL API may be accessed:

`gateway/.env`

```
NODE_ENV=development
PORT=4000
```

Next, we'll configure a basic, non-federated Apollo/Express server in the `index.js` file first, which will ultimately be used for the gateway. We'll start by importing the required modules, getting the `PORT` environment variable value, and creating a new Express app inside of it:

`gateway/src/index.js`

```
import http from "http";

import { ApolloServer, gql } from "apollo-server-express";
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";
import express from "express";

const port = process.env.PORT;
const app = express();
const httpServer = http.createServer(app);

// We'll create an Apollo Server here next...
```

In the code above, note that the `ApolloServerPluginDrainHttpServer` plugin will allow us gracefully shut down Apollo Server when using one of its Node.js integrations, such as Express. For initial testing purposes, we'll add a single `hello` field to the root `Query` type and create a map of resolver functions for the schema:

`gateway/src/index.js`

```
// ...

const typeDefs = gql`  
  type Query {  
    hello: String
```

```
    }
};

const resolvers = {
  Query: {
    hello() {
      return "world";
    }
  }
};
```

Next, we'll instantiate a new `ApolloServer`, integrate it with the Express app defined above by adding it to the app's middleware, and then listen for connections on port `4000`:

`gateway/src/index.js`

```
// ...

const gateway = new ApolloServer({
  typeDefs,
  resolvers,
  plugins: [ApolloServerPluginDrainHttpServer({ httpServer })]
});

await gateway.start();
gateway.applyMiddleware({ app, path: "/" });

await new Promise(resolve => httpServer.listen({ port }, resolve));
console.log(`Gateway ready at http://localhost:${port}${gateway.graphqlPath}`);
});
```

Now we can start our server by running the following command:

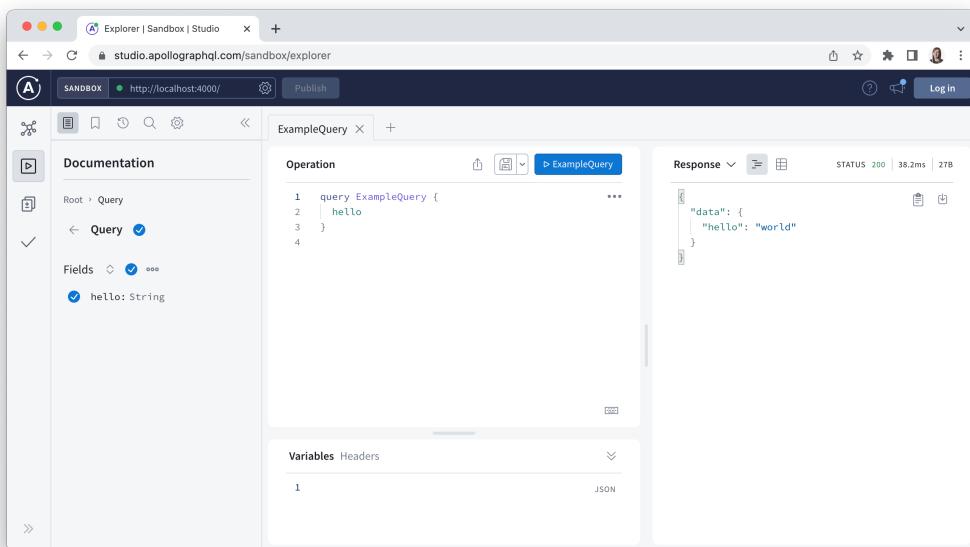
`gateway/`

```
npm run dev
```

We can use Apollo Sandbox Explorer to check if the GraphQL API is working as expected at this point. The Explorer IDE is readily available through a running Apollo Server and can be accessed in a browser via the same URL that we use for the GraphQL endpoint itself. In our case, this URL is `http://localhost:4000` and once you reach the landing page you can access the Explorer user

interface by clicking the “Query your server” button. If you’re new to Explorer but have used GraphQL Playground or GraphiQL in the past, then you should feel right at home.

Once you’ve opened Explorer you should be able to see the `hello` field that we added to the root `Query` type along with an example operation. Click the blue button to submit the operation and confirm that you can see the following response:



We are currently using Explorer inside of Apollo Sandbox, which means we don’t need to create an Apollo Studio account to run operations against our schema. In Chapter 9, we will create an Apollo Studio account to configure [managed federation](#) for our GraphQL API, which will also provide access to the full-featured version of Explorer.

## Organize the Server Files

To set ourselves up for success later, we’ll do some file reorganization now because it will quickly become cumbersome to keep all of our gateway code inside of a single `index.js` file. Start by creating a `config` subdirectory in `gateway/src`, then create `apollo.js` and `app.js` files inside of `config`. We’ll move the following code from `index.js` to `apollo.js`, wrapping it in a function called `initGateway` and returning the `ApolloServer` object from it:

*gateway/src/config/apollo.js*

```
import { ApolloServer, gql } from "apollo-server-express";
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";

function initGateway(httpServer) {
  const typeDefs = gql` 
    type Query {
      hello: String
    }
  `;

  const resolvers = {
    Query: {
      hello() {
        return "world";
      }
    }
  };

  return new ApolloServer({
    typeDefs,
    resolvers,
    plugins: [ApolloServerPluginDrainHttpServer({ httpServer })]
  });
}

export default initGateway;
```

Now we'll move the `express` import and the `app` variable into `app.js`. We will also make `app` the default export from this file:

*gateway/src/config/app.js*

```
import express from "express";

const app = express();

export default app;
```

Next, we can import the new modules at the top of `index.js`, call the `initGateway` function, and assign its return value to `gateway` instead. The updated `index.js` file will look like this now:

gateway/src/index.js

```
import http from "http";

import app from "./config/app.js";
import initGateway from "./config/apollo.js";

const port = process.env.PORT;
const httpServer = http.createServer(app);
const gateway = initGateway(httpServer);

await gateway.start();
gateway.applyMiddleware({ app, path: "/" });

await new Promise(resolve => httpServer.listen({ port }, resolve));
console.log(`Gateway ready at http://localhost:${port}${gateway.graphqlPath}`);
);
```

And the current file structure in our project directory will now look like this:

```
gateway
├── node_modules/
│   └── ...
├── src/
│   ├── config
│   │   └── apollo.js
│   │   └── app.js
│   └── index.js
└── .env
└── package.json
└── package-lock.json
```

When we save the new files and the server restarts we should see that the GraphQL API works just as it did before.

## Why Use Apollo Federation?

Before we proceed with refactoring our code further to convert our existing Apollo Server into a gateway, it would be a good idea to pause and consider why we would want to do this in the first

place. We have a perfectly viable GraphQL API already up and running, so how do we know that using Apollo Federation now will be the right choice moving forward?

One of the most compelling reasons to create any kind of GraphQL API is that it allows us to expose a single API endpoint (with any number of data sources behind it) and query data in a way that reflects the natural relationships between the nodes in the graph. However, as we express more and more pieces of data and their relationships through types and fields, it doesn't take long for even a moderately complex GraphQL API to contain many different type definitions.

A large number of type definitions doesn't necessarily warrant a federated approach on its own, but there tends to be a positive correlation between the number of type definitions in a schema and the number of teams ultimately making contributions to that schema. And as more people contribute to the active evolution of the schema (that is one of the promises of GraphQL, after all!), a monolithic GraphQL API can quickly become a bottleneck for teams that need to ship updates to type definitions that are relevant to the services they maintain.

Alternatively, multiple GraphQL APIs may simultaneously spring up within an organization, leading to duplication of effort to maintain them and inevitable type definition inconsistencies that makes it challenging for clients to query any GraphQL endpoint as a definitive source of truth for all of the data they require. In the end, both the monolithic and multiple GraphQL API options have obvious shortcomings, especially when used with well-worn approaches to distributed application development such as microservice architectures.

Ideally, where multi-team collaboration is required, we would break up a GraphQL schema and assign ownership of type definitions to teams that already manage services and data sources that will be responsible for resolving those fields. Within their own *subgraphs*, teams should also be able to use types defined in other subgraphs as output types for fields and even augment those types with additional fields where needed. Lastly, these implementation details should be opaque to client developers. In other words, clients should be able to enjoy one of the key advantages of GraphQL and query a single endpoint as the source of truth for all of the data they require to render user interfaces. This is precisely what Apollo Federation allows us to do.

So in short, there are two main architectural components specific to Apollo Federation:

- The *subgraph* services that each represent part of the overall graph
- A top-level *gateway* service whose API is composed of the subgraph schemas

With these concepts in mind, we're almost ready to set up our first subgraph service.

### What about schema stitching?

Before Apollo Federation, there was an existing solution available to build these top-level gateway APIs with GraphQL called *schema stitching*. However, with schema stitching we may find ourselves wrestling with where to draw boundaries between services. Splitting services by type may seem intuitive at first but can get messy as we try to express complex relation-

ships between the different types across services. It doesn't take long for numerous "ID" fields to begin showing up in a service's schema that look more like foreign key references than a real connection to a type from the other service. When fields such as this appear an API begins to lose some of the expressiveness at the heart of GraphQL.

Additionally, schema stitching can require significant boilerplate code at the gateway level to effectively consolidate the graph, and that introduces a critical point of possible failure in an application's architecture.

Schema stitching was a step in the right direction for supporting distributed applications powered by multiple GraphQL APIs, but Apollo Federation presents a next-generation approach that allows us to declaratively define the gateway API in a way that cuts down on boilerplate and also allows a more natural separation of concerns between services.

## But First, More Refactoring

Before we attempt to build a subgraph schema for use in the gateway, we'll need to do more refactoring to tidy up our Apollo Server set-up in `apollo.js`. The first subgraph we create will handle user account data from Auth0. To begin, we'll need to create an `accounts` subdirectory inside of the top-level project directory (it will be a sibling to `gateway`).

It's worth noting here that the gateway and subgraph services we create for the Marked application would likely be independent projects with different teams maintaining each of them in separate repositories. However, for simplicity's sake, we'll organize them in a single directory for all of our project's code.

We'll create a `src` directory inside of `accounts`, and then an `index.js` file and `graphql` subdirectory inside of `src`. Next, we'll add `resolvers.js` and `schema.graphql` files inside of `accounts/src/graphql`. Our next step will be to move the root `Query` type and its single `hello` field into the new `schema.graphql` file. We are using a `.graphql` file extension and writing our type definitions using GraphQL's Schema Definition Language (SDL) so there's no need to wrap this code in the `gql` template literal tag or define a `typeDefs` variable here as we would if we were using a `.js` file for the type definitions instead.

However, because we will be using the second version of the [the Federation subgraph specification](#) in this project (also known as "Federation 2"), we must include a special `extend schema` definition at the top of the file:

accounts/src/graphql/schema.graphql

```
extend schema
@link(url: "https://specs.apollo.dev/federation/v2.0", import: ["@key"])

type Query {
  hello: String
}
```

Without the `extend schema` definition in this file, Apollo Federation's composition algorithm would assume this subgraph uses Federation 1 semantics. While both Federation 1 and Federation 2 subgraphs can be composed together, only subgraphs that include the previous definition can make use of Federation 2 semantics. The `@link` directive's `import` argument indicates which of [Apollo Federation's directives](#) will be used in the subgraph schema. For the accounts services, we will only need to import the `@key` directive, but we will use other Federation 2 directives for subgraphs in subsequent chapters.

Now we'll move the `resolvers` constant from `apollo.js` into the new `resolvers.js` file and make `resolvers` the default export from it:

accounts/src/graphql/resolvers.js

```
const resolvers = {
  Query: {
    hello() {
      return "world";
    }
  }
};

export default resolvers;
```

We'll then create a `server` variable in the accounts service's `index.js` file and instantiate a new `ApolloServer` here, passing in `typeDefs` and `resolvers` as we did previously in the gateway:

accounts/src/index.js

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

You may have noticed that we instantiate an `ApolloServer` in the accounts service now but we haven't yet imported this module at the top of the `index.js` file. We'll need to create a

package.json file for the accounts service too and install this dependency. We won't need to add Express middleware in this service, so the basic `apollo-server` package will be sufficient. We can also omit the `ApolloServerPluginDrainHttpServer` plugin here because this is handled under the hood when using Apollo Server without the explicit Express integration. We will need to install `graphql`, `dotenv`, and `nodemon` here as well. Run the following commands directly inside the `accounts` directory to do that:

```
accounts/
```

```
npm init --yes
npm i apollo-server@3.7.0 dotenv@16.0.0 graphql@16.5.0
npm i -D nodemon@2.0.15
```

We'll update the new `package.json` file to support ES modules and include a script to start the server with `nodemon`, also watching for changes to `.graphql` files this time:

```
accounts/package.json
```

```
{
  // ...
  "type": "module",
  "scripts": {
    "dev": "nodemon -r dotenv/config -e env,graphql,js ./src/index.js"
  },
  // ...
}
```

Now we can add the import for it at the top of `index.js`:

```
accounts/src/index.js
```

```
import { ApolloServer, gql } from "apollo-server";

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

Each subgraph service will have a unique Node.js process and expose its portion of the API on a unique port. Let's create a dedicated `.env` file in `accounts` and set its `NODE_ENV` and also set its `PORT` value to be a different port number than the gateway:

accounts/.env

```
NODE_ENV=development
PORT=4001
```

Over in the accounts service's `index.js` file, we'll import the resolvers and define a port constant using the environment variable. Unlike before when we defined the type definitions as a string in a `.js` file, we put our type definitions in a `.graphql` file here so we have to read the contents of that file as a string and pass it into the `gql` function to parse it into a standard GraphQL AST. Note that we must also manually define the `__dirname` constant when using ES modules, so we will import some additional built-in modules to assist with that:

accounts/src/index.js

```
import { dirname, resolve } from "path";
import { fileURLToPath } from "url";
import { readFileSync } from "fs";

import { ApolloServer, gql } from "apollo-server";

import resolvers from "./graphql/resolvers.js";

const __dirname = dirname(fileURLToPath(import.meta.url));
const port = process.env.PORT;

const typeDefs = gql(
  readFileSync(resolve(__dirname, "./graphql/schema.graphql"), "utf-8")
);

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

With that code in place, we can start the accounts service's server by calling the `server` object's `listen` method and logging a success message to the console:

accounts/src/index.js

```
// ...

const { url } = await server.listen({ port });
console.log(`Accounts service ready at ${url}`);
```

Lastly, we'll do some final clean-up by removing the gql import from `apollo-server-express` in the `apollo.js` file because we no longer need it here. We can also remove the typeDefs and resolvers that were passed directly into this ApolloServer before. The `apollo.js` file will currently look like this:

`gateway/config/apollo.js`

```
import { ApolloServer } from "apollo-server-express";
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";

function initGateway(httpServer) {
  return new ApolloServer({
    plugins: [ApolloServerPluginDrainHttpServer({ httpServer })]
  });
}

export default initGateway;
```

Unfortunately, our gateway server can't be restarted quite yet because the `initGateway` function no longer returns a valid `ApolloServer` object. To address this error, we will finally convert this Apollo Server into a gateway in the next section.

## Compose a Subgraph into the Gateway

It's time to finish transforming the accounts service's GraphQL API into a subgraph that can be composed into a schema that will be used by the Apollo Gateway. After that, we'll set up the gateway-level GraphQL API so we can get that server back up and running. To begin, we'll install the `@apollo/subgraph` package in the `accounts` directory:

`accounts/`

```
npm i @apollo/subgraph@2.0.3
```

All of the code in this project complies with v2 of the Apollo Federation subgraph specification, so be sure to install 2.x versions of the `@apollo/subgraph` and `@apollo/gateway` packages as you work through this book.

To make the accounts service's schema federation-ready, we have to make some additions to our schema so that the gateway API can execute entity-related queries against it. Specifically, we need to add the following types, fields, and directives to the schema:

```
scalar _Any
scalar _FieldSet

union _Entity = # a union of all types that use the @key directive

type _Service {
  sdl: String
}

extend type Query {
  _entities(representations: [_Any!]!): [_Entity]!
  _service: _Service!
}

directive @key(fields: FieldSet!, resolvable: Boolean = true) repeatable on
  OBJECT | INTERFACE
directive @requires(fields: FieldSet!) on FIELD_DEFINITION
directive @provides(fields: FieldSet!) on FIELD_DEFINITION
directive @external on OBJECT | FIELD_DEFINITION
directive @shareable on FIELD_DEFINITION | OBJECT
directive @extends on OBJECT | INTERFACE
directive @override(from: String!) on FIELD_DEFINITION
directive @inaccessible on
  | FIELD_DEFINITION
  | OBJECT
  | INTERFACE
  | UNION
  | ENUM
  | ENUM_VALUE
  | SCALAR
  | INPUT_OBJECT
  | INPUT_FIELD_DEFINITION
  | ARGUMENT_DEFINITION
```

You can learn more about how these types, fields, and directives support entity resolution in the [Apollo Federation specification](#).

By following this specification and adding support for these definitions to a schema, any GraphQL server in any language can also provide support for Apollo Federation.

Luckily, we don't have to add these definitions to any of our subgraphs manually because the `@apollo/subgraph` package exposes a function called `buildSubgraphSchema` that does this work for us. In the accounts service's `index.js` file we'll import `buildSubgraphSchema` from `@apollo/subgraph`:

`accounts/src/index.js`

```
// ...  
  
import { ApolloServer, gql } from "apollo-server";  
import { buildSubgraphSchema } from "@apollo/subgraph";  
  
// ...
```

And we also need to change how we instantiate the `ApolloServer` in that file by calling `buildSubgraphSchema`, passing the `typeDefs` and `resolvers` directly into that function as arguments, and then setting the return value for the `schema` option instead:

`accounts/src/index.js`

```
// ...  
  
const server = new ApolloServer({  
  schema: buildSubgraphSchema({ typeDefs, resolvers })  
});  
  
const { url } = await server.listen({ port });  
console.log(`Accounts service ready at ${url}`);
```

The accounts service has finally reached a point where we can successfully start the server. From the `accounts` directory, run the following command:

`accounts/`

```
npm run dev
```

Because each subgraph is a self-contained GraphQL API, we can view and run queries against it in Explorer just as we did with the gateway. If you navigate to <http://localhost:4001> in your browser and launch Explorer, then you'll see the `hello` field on the root `Query` type, along with the `_service` field that was added when we decorated our subgraph schema using the `buildSubgraphSchema` function (note that the `_entities` field won't be added to the schema until we define an entity type later in the chapter).

Let's try running the following operation in Explorer:

### GraphQL Query

```
query {
  _service {
    sdl
  }
}
```

The string value of the `sdl` field in the response will be quite long, but if we take a close look then we should see that the SDL string for the subgraph includes the `extend schema` and `Query` type definitions as expected, and it also includes definitions from the Federation 2 specification.

Querying a subgraph endpoint as we just did can be useful in a development environment. However, it's a best practice in production environments to restrict the outside world from sending requests directly to subgraphs and only allow the gateway to run operations against these endpoints.

Now that the accounts service's subgraph is ready we can create an initial *supergraph* schema from it to pass into an `ApolloGateway`. A supergraph schema is a GraphQL schema that has been composed of one or more subgraphs. This schema won't be exposed directly to clients but will include some additional metadata that allows an Apollo Gateway to know what subgraphs can resolve which fields while also allowing it to generate the client-friendly schema that we would expect to be served by the gateway API.

There are a few different ways that we can generate a supergraph schema's SDL for the gateway. The approach that we'll take now is to instantiate a new `IntrospectAndCompose` object (provided by `@apollo/gateway`) and pass it into an `ApolloGateway` object as the `supergraphSdl` option. To do that, we'll need to install the Apollo Gateway package in the `gateway` directory:

`gateway/`

```
npm i @apollo/gateway@2.0.3
```

We will also set an environment variable for the `ACCOUNTS_ENDPOINT` in the gateway's `.env` file:

`gateway/.env`

```
NODE_ENV=development
PORT=4000

ACCOUNTS_ENDPOINT=http://localhost:4001
```

We will then configure the `ApolloGateway` object as follows and set it as the `gateway` option in the `ApolloServer` object:

`gateway/src/config/apollo.js`

```
import { ApolloGateway, IntrospectAndCompose } from "@apollo/gateway";
import { ApolloServer } from "apollo-server-express";
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";

function initGateway(httpServer) {
  const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: "accounts", url: process.env.ACCOUNTS_ENDPOINT }
      ],
      pollIntervalInMs: 1000
    })
  });

  return new ApolloServer({
    gateway,
    plugins: [ApolloServerPluginDrainHttpServer({ httpServer })]
  });
}

export default initGateway;
```

By providing a list of subgraph endpoints to `IntrospectAndCompose`, the gateway can introspect these endpoints—much like we did using the `_service` field above—and compose the supergraph SDL at runtime. We also set a poll interval of 1000 milliseconds so that the gateway will fetch updated SDLs from the subgraphs automatically as we develop them. Note that the `IntrospectAndCompose` option is suitable for development environments, but a better approach for production environments is to use *managed federation* with Apollo Studio, as we will see in Chapter 9.

With that code in place, we're ready to start up the gateway. Confirm that the accounts service is still up and running on <http://localhost:4001> and then open a second terminal tab or window and run the following command in the `gateway` directory:

`gateway/`

```
npm run dev
```

Congratulations! You just created your first federated GraphQL API. If we revisit <http://localhost:4000> in a browser, then we will be able to see the gateway running with the single `hello` field available on the root `Query` type, just as it was with our previous non-federated implementation.

## The First Entity

Allowing individual teams to each manage a subsection of type definitions, expose those schemas via dedicated GraphQL APIs, and then compose them into a larger GraphQL API for client applications is pretty powerful on its own, but we have yet to take advantage of *entities*, which are one of Apollo Federation's true superpowers.

Entities provide connection points between subgraphs. More specifically, an entity is an Object or Interface type that we define in one subgraph and then reference and extend in other subgraphs. Ultimately, entities allow us to draw subgraph boundaries based on *separation of concerns* and continue to express natural type and field relationships in a subgraph schema even when certain types are owned by other subgraphs.

Before we begin integrating Auth0 to support the accounts service, we'll define an `Account` Object type as an entity and create a resolver for it. We'll also add a new field on the root `Query` type that's more relevant to the accounts service than the `hello` field.

First, we'll update the accounts service's schema:

`accounts/src/graphql/schema.graphql`

```
extend schema
@link(url: "https://specs.apollo.dev/federation/v2.0", import: ["@key"])

type Account {
  id: ID!
  email: String!
}

type Query {
  viewer: Account
}
```

To make the `Account` type an entity we must add an `@key` directive to its definition. The `@key` directive accepts a `fields` argument where we indicate what field or fields on the `Account` type can be used to uniquely identify a given account. You can think of it as a primary key for data that the type represents. In this case, it's the `id` field:

accounts/src/graphql/schema.graphql

```
# ...

type Account @key(fields: "id") {
  id: ID!
  email: String!
}

# ...
```

By declaring the `id` as the `@key` field for `Account`, other subgraphs will now be able to use this entity type and the gateway will know that an account can be resolved across subgraphs if the referencing subgraph knows the account's unique ID.

Now we'll update our resolvers to return the new `Account` entity for the `viewer` field. First, we'll add a `const` with some mocked data for testing purposes at the top of the file:

accounts/src/graphql/resolvers.js

```
const accounts = [
  {
    id: "1",
    email: "marked@mandiwise.com"
  }
];

// ...
```

And then we'll update our `resolvers` object to resolve data for the `Account` type and `viewer` field on the root `Query` type:

accounts/src/graphql/resolvers.js

```
// ...

const resolvers = {
  Account: {
    __resolveReference(reference) {
      return accounts.find(account => account.id === reference.id);
    }
  },
}
```

```
Query: {
  viewer() {
    return accounts[0];
  }
};

export default resolvers;
```

Above, we have our first encounter with a *reference resolver*. A reference resolver is a special concept in Apollo Federation and it's the way that we resolve an instance of an entity based on its primary key alone. To further clarify why we need a reference resolver, it may be helpful to imagine what will go on behind the scenes here. When a different subgraph tries to reference Account it must pass some data back to the gateway to identify a given account and the gateway will in turn supply that data to the accounts service inside of the `reference` object parameter of the `__resolveReference` resolver. We can access the ID of an account inside of the `reference` object and use it to fetch all of the required data for the account from there.

We'll also eventually see how we need to write resolvers in a special way for any Account fields that are extended by other subgraph schemas. These two kinds of resolvers are how we connect entities with other types and fields together through the gateway. But more on that to come later!

It's also worth noting that a reference resolver has the following parameters available:

```
__resolveReference(reference, context, info) { /* ... */ }
```

We only need `reference` for our purposes right now, but just like regular resolvers, we can also access the `context` object (which we'll make use of shortly) and the `info` object (which contains advanced information about the field and operation).

Now we can try out our new `viewer` query against the gateway endpoint in Explorer:

*GraphQL Query*

```
query Viewer {
  viewer {
    id
    email
  }
}
```

*API Response*

```
{  
  "data": {  
    "viewer": {  
      "id": "1",  
      "email": "marked@mandiwise.com"  
    }  
  }  
}
```

## Optional: Supergraph Composition with the Rover CLI

Using `IntrospectAndCompose` is just one way to provide a supergraph SDL to a gateway. We can also use Apollo's Rover CLI to compose the supergraph SDL manually as a static artifact. To try this method out, you'll need to install the Rover binary following the instructions in the "Required Software" section of the Preface.

The Rover CLI exposes a `rover supergraph compose` command that will compose a schema based on a YAML configuration file. Add a `supergraph.yaml` file to the `gateway/src` directory with the following code in it:

```
gateway/src/supergraph.yaml
```

```
subgraphs:  
accounts:  
  routing_url: http://localhost:4001  
  schema:  
    subgraph_url: http://localhost:4001
```

In the code above, the `routing_url` option tells the gateway API where to send requests that will be fulfilled by the accounts service when resolving an operation and the `subgraph_url` option indicates where the accounts service's schema can be introspected.

With the accounts service running, we can run the following command from the `gateway` directory and see the supergraph schema output in the terminal in its SDL representation:

```
gateway/
```

```
rover supergraph compose --config src/supergraph.yaml
```

We should see the following output:

```
schema
  @link(url: "https://specs.apollo.dev/link/v1.0")
  @link(url: "https://specs.apollo.dev/join/v0.2", for: EXECUTION)
{
  query: Query
}

directive @join__field(graph: join__Graph!, requires: join__FieldSet,
  provides: join__FieldSet, type: String, external: Boolean, override:
  String, usedOverridden: Boolean) repeatable on FIELD_DEFINITION |
  INPUT_FIELD_DEFINITION

directive @join__graph(name: String!, url: String!) on ENUM_VALUE

directive @join__implements(graph: join__Graph!, interface: String!)
  repeatable on OBJECT | INTERFACE

directive @join__type(graph: join__Graph!, key: join__FieldSet, extension:
  Boolean! = false, resolvable: Boolean! = true) repeatable on OBJECT |
  INTERFACE | UNION | ENUM | INPUT_OBJECT | SCALAR

directive @link(url: String, as: String, for: link__Purpose, import:
  [link__Import]) repeatable on SCHEMA

type Account
  @join__type(graph: ACCOUNTS, key: "id")
{
  id: ID!
  email: String!
}

scalar join__FieldSet

enum join__Graph {
  ACCOUNTS @join__graph(name: "accounts", url: "http://localhost:4001")
}

scalar link__Import

enum link__Purpose {
  """
```

```
 `SECURITY` features provide metadata necessary to securely resolve
fields.
"""
SECURITY
"""
`EXECUTION` features provide metadata necessary for operation execution.
"""
EXECUTION
}

type Query
  @join__type(graph: ACCOUNTS)
{
  viewer: Account
}
```

Again, what we see here isn't the schema that will be served to clients by the gateway, but rather a configuration for our gateway API that is expressed as a GraphQL schema. So in practice, there are three different kinds of schemas in use when using Apollo Federation:

- **Subgraph schemas:** These schemas are defined by individual services (where their fields are also resolved) and composed together for use in the gateway GraphQL API.
- **Supergraph schema:** This schema is used by an `ApolloGateway` to serve the client-facing schema and to route requests to subgraph services based on the fields in an operation.
- **API schema:** The composed, client-facing schema where federation-related implementation details in the subgraph and supergraph schemas have been removed.

We could write the output of the previous Rover command to a file and then read its contents and provide it directly to an `ApolloGateway` object as the `supergraphSdl` option, but we would also need to add some logic to re-fetch any subgraph schemas as they change. This is what `IntrospectAndCompose` handles in our development environment. In production, using a schema registry (such as the one in Apollo Studio) to track subgraph changes while the gateway polls the registry for updates is typically a better option than continually polling the subgraphs with `_service` queries. We'll explore managed federation with Apollo Studio further in Chapter 9.

## Summary

In this chapter, we have laid a solid foundation for the Marked GraphQL API. Our development environment is now configured and we have a gateway Apollo Server running with Express, as well as our first subgraph schema for the accounts service, which is running on a separate Node.js process from the gateway. In the next chapter, we'll continue to build out the accounts service by adding authentication with Auth0.

## Chapter 2

# Authentication and User Account Management with Auth0

In this chapter, we will:

- Sign up for Auth0 and use it to authenticate requests to a GraphQL API
- Add middleware to Express to verify JSON Web Tokens sent with incoming requests
- Install and configure an Auth0 client library for Node.js to make requests to Auth0's Management API
- Build out the Account type further by adding fields and some basic queries
- Add mutations for creating, updating, and deleting user accounts

## Sign Up for Auth0

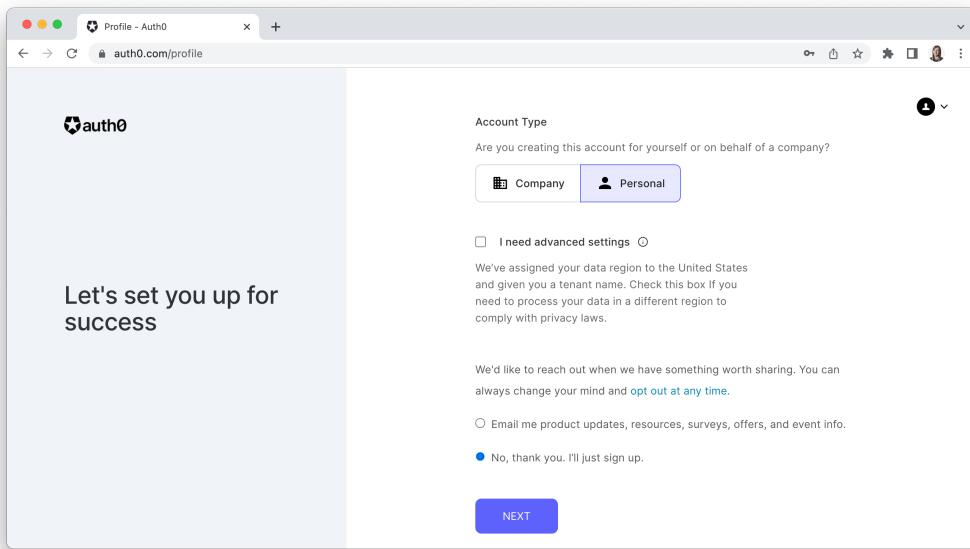
At the moment, our accounts service doesn't have anything to do with authenticating or authorizing user accounts yet. We need a way for users to sign up for a new account so they can access data that will only be available to authenticated users. We will use a third-party service called [Auth0](#) to begin building out these features.

Auth0 is an authentication and authorization platform for web and mobile apps. It facilitates many useful auth-related features such as social logins, single sign-on, and password-less authentication. You will need to sign up for an Auth0 account to use this service but it does have a free usage tier that covers all of the authentication and authorization features we'll need to build Marked right now.

You may be wondering, "why should we use a third-party service instead of building out authentication from scratch?" Some of the key architectural decisions you make when building an application rest on knowing when to not spend time reinventing solutions to challenging problems that have already been solved by other services or open-source libraries. For this reason,

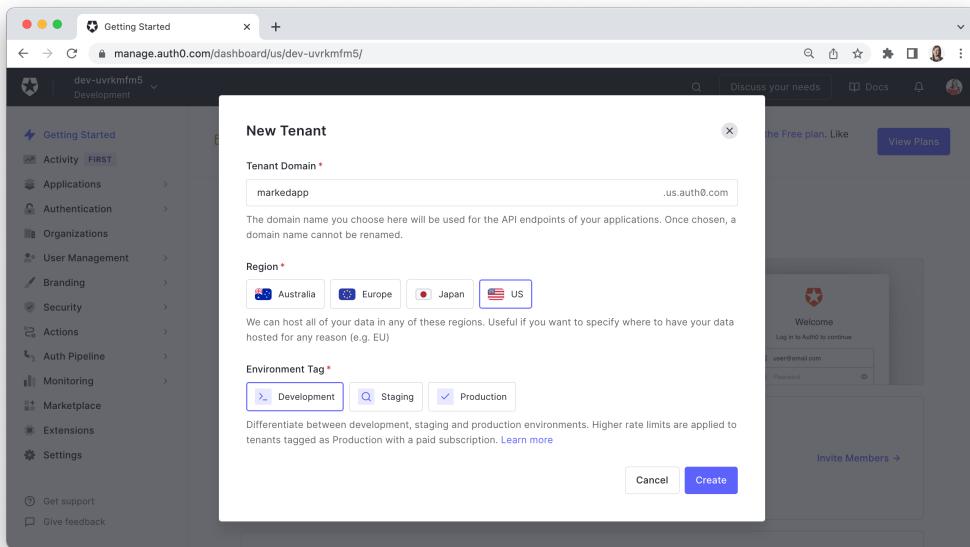
we will rely on Auth0 and its robust set of APIs to support the authentication requirements for this app and then focus our software development efforts on GraphQL-specific concerns instead.

To get started, you'll need to sign up for an Auth0 account. You can sign-up with a variety of different social logins or a username and password. Once you choose your preferred sign-up method you will be taken to the screen below:

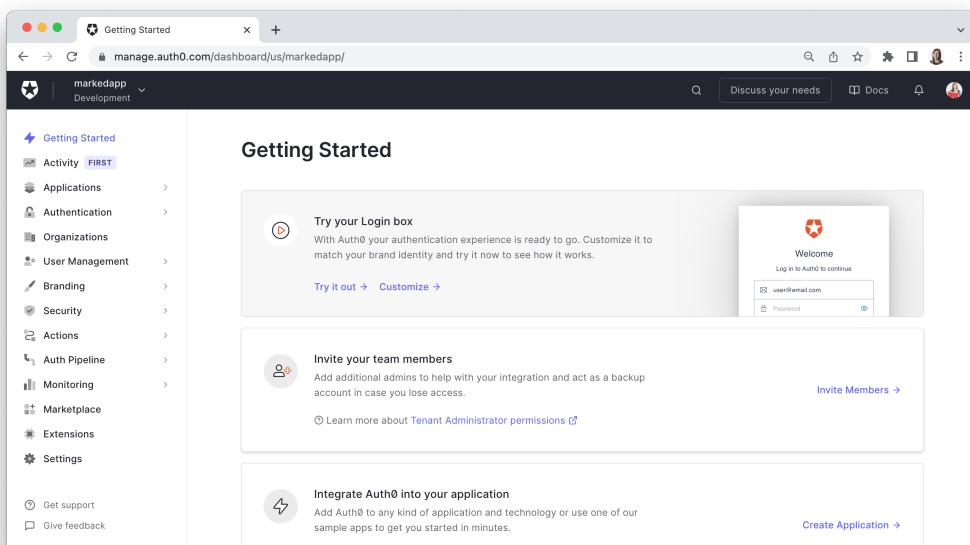


Once you complete all of the initial sign up steps, Auth0 will automatically create a default *tenant* for you and redirect you to its dashboard. A tenant is a logical unit of isolation within your new account. You can create more than one tenant per Auth0 account, which is helpful when managing different applications, APIs, and user data for development and production environments.

Tenant names must be unique, use lowercase letters only, and they cannot be changed after creation (so choose your tenant name wisely!). If you prefer to use a different tenant name, then you can create a new one by choosing “Create tenant” from the menu in the top left-hand corner of the dashboard and setting a domain, region, and environment tag for it:



If you created a new tenant, then the dashboard should look something like this. Notice that the tenant name has changed in the top left-hand corner of the dashboard now:



## What Are JSON Web Tokens and How Do They Work?

Apart from managing user account data, by integrating Auth0 into Marked our goal is to make sure that if someone requests protected information, then we know that they are who they say they are. In other words, we want to know that they are *authenticated*. In the next chapter, we'll use custom directives to verify that a user is *authorized* to see and change the specific data involved in an API request.

To achieve this end, Auth0 will issue a user an *access token* when they successfully authenticate with the app. Access tokens allow applications to make requests to an API on a user's behalf. A user's browser can send this token along with each subsequent request so that the user doesn't need to re-authenticate until the token expires.

Auth0 will issue the access token in the form of a *JSON Web Token* (JWT). A JWT conforms to an open standard that describes how information may be transmitted as a compact JSON object. JWTs consist of three distinct parts:

1. **Header:** Contains information about the token type and the algorithm used to sign the token (for example, RS256).
2. **Payload:** Contains claims about a particular entity, and these statements may have predefined meanings in the JWT specification (known as *registered claims*) or they can be defined by the JWT user (known as *public* or *private* claims).
3. **Signature:** Helps to verify that no information was changed during the token's transmission by hashing together the token header, its payload, and a secret.

To create the token, each of these parts is base64url-encoded (for compactness) and then concatenated together with a dot. A typical JWT may look like this:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIiOiIxMjM0NTY3ODkwIiwiaWF0IjoxNjTE2MjM5MDIyLCJuYW1lIjoiQm9iIFJvc3MiLCJwYWludGVyIjp0cnVlfQ.IAHZ341NCj8ggBRNjirYcLcV1nQUAaQt_P-FWJI-utW5nYhRj0EzFp_pyhz5JlyGMKveTJYxYF2_YuM7mSHBdbcUpoUi-Z8JdqdoCifuddIcOL3ZaLCQhkgl84gf_T_u77a9PLjuRhExkYFjd73CTsXQUJgjn8YnKC8263jVsYktN6N7iFcvo0gWyRwdX4xWmyiES3LJWjwPW8cWInJIcc6m3z488URHlvhlDshlOTJCWT0hrjVaG0yFhSzqtDl8dU5_E6InMd3qyuQNus5TBiMacLcuLcP8GzpEnp3yCeRUcuhpw8ltDK0GVsMMjVNBofWVEps5iDAG7zdY18GiJ5zvnLg
```

It's important to note that even though the JWT above may look encrypted at first glance it's just base64url-encoded, so all of the information inside can just as easily be decoded again. Similarly, the signature portion of the JWT only helps us ensure that its data hasn't been changed while in transit between the sender and receiver. The signature plays no role in actually encrypting the information contained within. For these reasons, it's important to not put any secret information inside of the JWT header or payload in plain text.

The header and payload sections of the above token would respectively decode to:

### JWT Header

```
{  
  "alg": "RS256",  
  "typ": "JWT"  
}
```

### JWT Payload

```
{  
  "sub": "1234567890",  
  "iat": 1516239022,  
  "name": "Bob Ross",  
  "painter": true  
}
```

In the token's payload, the `sub` and `iat` claims represent *registered* claims, where `sub` (short for “subject”) is a unique identifier for the object described by the token. The `iat` claim is the time at which the token was issued. These claims are a part of the JWT specification. However, `artist` and `painter` are *custom* claims added to token, and specifically, they would be considered *private* claims and meant for use in a closed network only. By contrast, a public claim is also user-defined but would be registered with the IANA JSON Web Token Claims registry or have a specially formatted name (such as a URI) to avoid naming collisions. There are no public claims in the token above.

You can experiment with encoding and decoding JWTs at <https://jwt.io>.

We now have some knowledge of what JWTs are and how they can help us send information between a client application and API to verify a user's identity. Our next step will be to figure out how to process incoming JWTs on the server when a client has made a request to our GraphQL API.

## Check JWTs with Express Middleware

We're going to set up some Express middleware to handle the JWTs that are sent with incoming requests to the gateway. Once the middleware is fully configured, we'll have to send an access token in the HTTP headers of the request when we include fields in an operation that are available to authenticated users only.

To configure the JWT middleware we'll need to install a few more npm packages, including:

- **express-jwt**: Middleware for validating JWTs and setting a `user` object inside the Express `req` object. The `user` object will contain the decoded token payload so that it can be used for access control.
- **jwks-rsa**: A library to retrieve RSA signing keys from a *JSON Web Key Set* (JWKS) endpoint. A JWKS is a set of keys containing the public keys that should be used to verify any JWT issued by the authorization server. Auth0 will provide the JWKS endpoint for us. We need this package to validate the signature of the JWT Auth0 issues because this token will be signed with a public/private key pair using RS256, rather than being signed symmetrically with a shared secret.

A deeper discussion about the use of various signing algorithms to sign JWTs is outside of the scope of this book, but you can [read more about RS256 and JWKS on the Auth0 blog](#).

We'll install both of these packages in `gateway` now:

`gateway/`

```
npm i express-jwt@6.1.1 jwks-rsa@2.0.5
```

We need to create some Auth0-specific environment variables to use with the JWT middleware:

`gateway/.env`

```
AUTH0_AUDIENCE=http://localhost:4000/
AUTH0_ISSUER=https://markedapp.us.auth0.com/
#
# ...
```

The `AUTH0_AUDIENCE` is the endpoint of our GraphQL API and the `AUTH0_ISSUER` is the unique Auth0 domain for the tenant. Please note that your Auth0 domain will be different from the one above. Your domain will be `YOUR-TENANT-NAME.auth0.com` (and use a region-specific subdomain if you picked a non-US region for your tenant), so be sure to use the correct value for the `AUTH0_ISSUER` variable. Now we'll update `app.js` with some new imports:

`gateway/src/config/app.js`

```
import express from "express";
import jwt from "express-jwt";
import jwksClient from "jwks-rsa";

// ...
```

Next, we'll set up the JWT-checking middleware for our Express app:

*gateway/src/config/app.js*

```
// ...  
  
const app = express();  
  
const jwtCheck = jwt({  
  secret: jwksClient.expressJwtSecret({  
    cache: true,  
    rateLimit: true,  
    jwksRequestsPerMinute: 5,  
    jwksUri: `${process.env.AUTH0_ISSUER}.well-known/jwks.json`  
  }),  
  audience: process.env.AUTH0_AUDIENCE,  
  issuer: process.env.AUTH0_ISSUER,  
  algorithms: ["RS256"],  
  credentialsRequired: false  
});  
  
export default app;
```

Briefly stated, the code above checks the validity of the access token included in the headers of a user's request. Less simply, under the hood, the `express-jwt` package will decode the token and then pass the request, the header, and the payload to `jwks-rsa`. Next, `jwks-rsa` will get the signing keys from the JWKS endpoint and check if one of the keys matches the `kid` in the header of the incoming JWT. If there's no matching key, then an error will be thrown, but if there is a match, then `jwks-rsa` will hand the correct signing key back to `express-jwt`. The `express-jwt` package will then validate the signature of the token and other details like expiration time, audience, and issuer.

You may find it strange that we set `credentialsRequired` to `false`, but we do so because we still want to provide some amount of access to unauthenticated users. If we set this value to `true` we wouldn't even be able to access the Explorer user interface in our development environment because the unauthenticated introspection query would fail. When we add authorization in Chapter 3 we will instead require users to provide a valid JWT when making requests for certain fields only.

The last thing to note about our JWT middleware configuration is that we set both `cache` and `rateLimit` to `true`. Doing so helps us avoid exceeding Auth0's rate limit for the JWKS endpoint. We can now add the JWT-checking middleware to our Express app:

gateway/src/config/app.js

```
// ...  
  
app.use(jwtCheck, (err, req, res, next) => {  
  if (err.code === "invalid_token") {  
    return next();  
  }  
  return next(err);  
});  
  
export default app;
```

You'll notice that we've added some custom logic to this middleware to manage authorized access to the Express application. We must do this because by default `express-jwt` will throw an error if a token is invalid, even with `credentialsRequired` set to `false`. We don't want Express to crash if the user submits an invalid token (for instance, if the token has exceeded its expiration time), so we must do some extra error handling here.

Lastly, we'll head over to `apollo.js` and add the value of the Express `req.user` object to the `context` in the Apollo Server. When setting the `context` property, we can either explicitly set it as an object or as a function that returns an object. For our purposes, we'll need to use the function option so we can access the `req` object from its parameter:

gateway/src/config/apollo.js

```
// ...  
  
function initGateway(httpServer) {  
  // ...  
  
  return new ApolloServer({  
    gateway,  
    plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],  
    context: ({ req }) => {  
      const user = req.user || null;  
      return { user };  
    }  
  });  
}  
  
export default initGateway;
```

The `context` stores data that is shared by all resolver functions, such as information about a currently authenticated user. This data can then be used inside resolver functions to modify what data is returned from a field or throw an error under certain conditions, as we'll see later on.

From the parameter Apollo makes available to this function, we destructure the `req` property to then access the `user` property that our JWT middleware has added. We then conditionally set information about the current user in the Apollo Server `context` if that information is available.

Back over in the accounts service's `resolvers.js` file, let's update the `viewer` query to destructure the `user` from the `context` object, which is the third parameter available in the resolver function:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Query: {
    viewer(parent, args, { user }) {
      console.log(user);
      return accounts[0];
    }
  }
};

export default resolvers;
```

If we try running the `viewer` query from Explorer now, then we'll see that we log a value of `undefined`. This may seem strange because—while we wouldn't expect to see any information about a current user because we haven't sent an access token along with our request—we would at least expect to see `null` based on how we set the `user` property in the `context` object.

Ultimately, this happens because we've set the `user` on the `context` at the gateway level of our API, rather than at the individual subgraph level. We'll need to write some extra code to pass that context along from the gateway to subgraphs so we can use it in the service's resolvers.

## Pass Context from the Gateway and the Accounts Service

For our gateway to share context with subgraph services, we'll need to do two things. First, we're going to need to do a little extra work when we instantiate the `ApolloGateway` to grab the gateway's `user` context and pass it as part of the headers in the HTTP request to the underlying service. Second, we'll need to use that new header to then set the context within the accounts

service so we can access it in the `viewer` resolver. To do these two things, we'll start by updating `apollo.js` with a new import from `@apollo/gateway` called `RemoteGraphQLDataSource`:

`gateway/src/config/apollo.js`

```
import {
  ApolloGateway,
  IntrospectAndCompose,
  RemoteGraphQLDataSource
} from "@apollo/gateway";
import { ApolloServer } from "apollo-server-express";
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";

// ...
```

By instantiating a new `RemoteGraphQLDataSource`, we can pass additional data from the gateway via HTTP headers in its requests to the subgraphs as it executes a query plan. Specifically, we will need to return the `RemoteGraphQLDataSource` object from the `buildService` option in the `gateway`. The `buildService` function is called for each request the gateway makes to a subgraph to resolve data for a given operation. This function has a single parameter which is an object representing the subgraph that the gateway is querying.

To forward the authenticated user's decoded and validated JWT, we'll destructure the subgraph's `url` property from the parameter and instantiate a new `RemoteGraphQLDataSource` by setting its `url` property to destructured subgraph URL and using its `willSendRequest` method to add a `user` header to the request as follows:

`gateway/src/config/apollo.js`

```
// ...

function initGateway(httpServer) {
  const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
      // ...
    }),
    buildService({ url }) {
      return new RemoteGraphQLDataSource({
        url,
        willSendRequest({ request, context }) {
          request.http.headers.set(
            "user",
            context.user ? JSON.stringify(context.user) : null
        }
      })
    }
  })
  return gateway
}
```

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

// ...

}

export default initGateway;
```

In the code above, we can see that there is an object parameter available to `willSendRequest`. From it, we destructure its `request` and `context` properties—`context` being the context set in the gateway and `request` being the request that will be sent to a subgraph.

From there, we call `request.http.headers.set` and pass in `user` as the name of the header we want to set with the JSON-stringified `user` object from the gateway as its value. Alternatively, we'll set it to `null` if it has not been added to the gateway context. After that, we'll need to retrieve this header from the request once it reaches the accounts service to explicitly set the `user` context for this subgraph's `ApolloServer`:

*accounts/src/index.js*

```
// ...

const server = new ApolloServer({
  schema: buildSubgraphSchema({ typeDefs, resolvers }),
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user };
}
);

// ...
```

When we run the `viewer` query now we can see that `null` is logged as expected.

### Important caveat alert!

We decided to validate the JWT at the gateway and then pass the user data to the accounts service. The accounts service will not take any action to revalidate the original token and

will later handle field-level authorization based on the user data provided by the gateway and assume that it has not been tampered with in transit.

Doing so will simplify our project for instructional purposes, but in the real world, this approach would necessitate that, at a minimum, the subgraph endpoints are inaccessible to the outside world and that traffic between the gateway and subgraphs is encrypted.

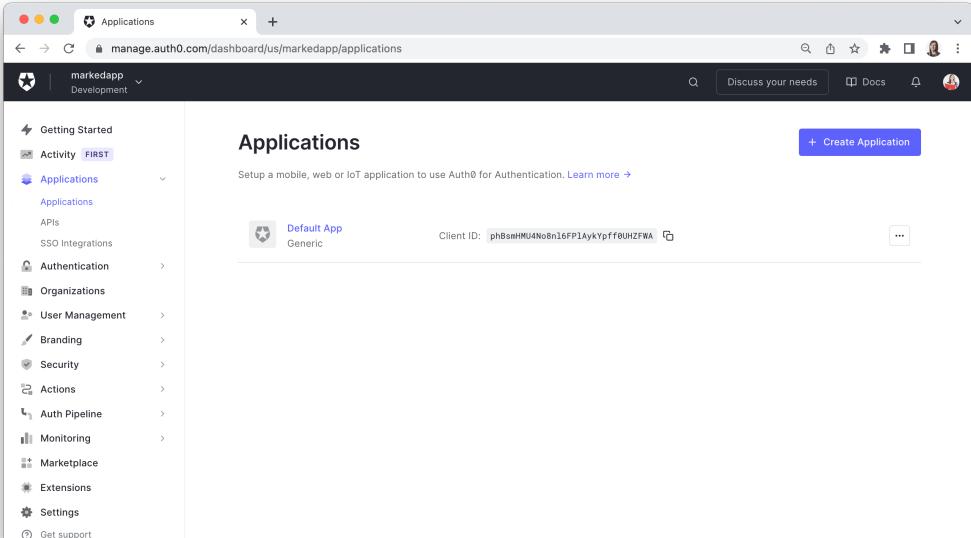
Alternatively, the original token could be forwarded from the gateway and validated in each subgraph. There would be a performance trade-off for this approach and you may want to ensure that the token is only validated once per operation for particularly complex operations that require multiple trips to a single subgraph.

## Create Auth0 Applications, APIs, and Users

Now we'll need to set up four more things to authenticate users with Auth0 and send a real access token along with our API requests:

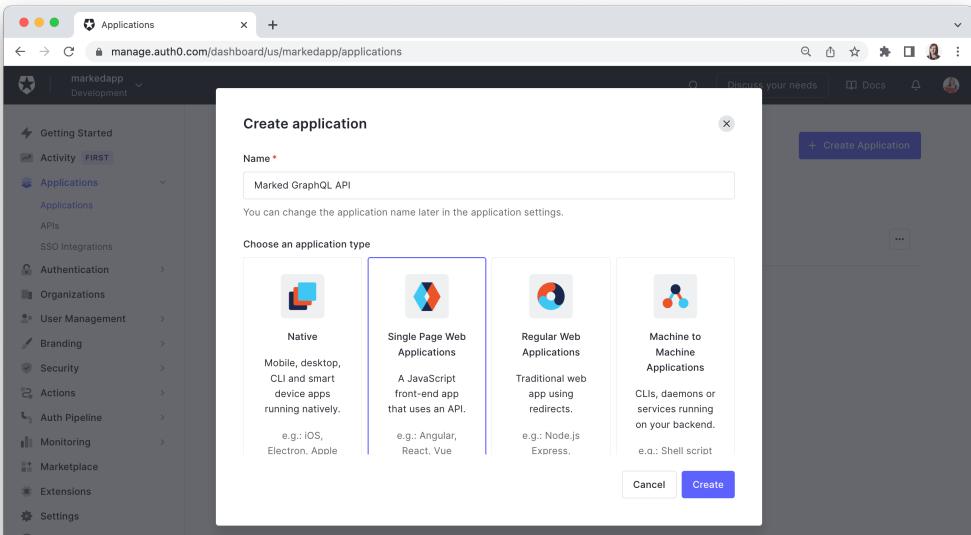
1. A new *application* in our Auth0 account to use with Explorer and the back-end application
2. A new *API* in our Auth0 account to use with Explorer
3. An initial Marked user set up in Auth0 so we can later validate their access token with the Express middleware
4. A script to send a request with information about the application, API, and user we just created to obtain an access token that we can use when testing operations in Explorer that require authentication

To authenticate requests from Explorer we must first register it as an application in Auth0. From the Applications page, click the “Create Application” button:



The screenshot shows the Auth0 dashboard with the URL `manage.auth0.com/dashboard/us/markedad/applications`. On the left, a sidebar menu is open under the 'Applications' section, listing various categories like 'Authentication', 'User Management', 'Branding', etc. The main content area is titled 'Applications' and contains a single entry for a 'Default App' with a generic icon. The Client ID is listed as `phBsmHRU4Nobn16fPlAykYptfBUHZFWA`. A blue button labeled '+ Create Application' is visible in the top right corner.

We'll give our application a descriptive name and then select the “Single Page Web Applications” type and click the “Create” button:



The screenshot shows the 'Create application' dialog box overlying the dashboard. The dialog has a title 'Create application' and a 'Name \*' field containing 'Marked GraphQL API'. Below it, a note says 'You can change the application name later in the application settings.' The 'Choose an application type' section contains four options: 'Native', 'Single Page Web Applications' (which is selected and highlighted with a blue border), 'Regular Web Applications', and 'Machine to Machine Applications'. Each option has a description and examples. At the bottom right of the dialog are 'Cancel' and 'Create' buttons.

Even though we'll use this application with Explorer, it's the option that is best suited for our purposes. Once the application is created and we jump to the "Settings" tab:

The screenshot shows the 'Application Details' section of the Auth0 dashboard. On the left, a sidebar lists various application settings like 'Getting Started', 'Activity', 'Applications', 'APIs', 'SSO Integrations', 'Authentication', 'Organizations', 'User Management', 'Branding', 'Security', 'Actions', 'Auth Pipeline', 'Monitoring', 'Marketplace', 'Extensions', 'Settings', and 'Get support'. The main content area is titled 'Marked GraphQL API' and describes it as a 'Single Page Application' with a 'Client ID' of '13sVZrbGp8ySe1sHYg0d44D9xxVGMI3F'. Below this, there are tabs for 'Quick Start', 'Settings' (which is selected), 'Addons', 'Connections', and 'Organizations'. The 'Settings' tab displays 'Basic Information' fields: 'Name' (set to 'Marked GraphQL API'), 'Domain' (set to 'markedapp.us.auth0.com'), 'Client ID' (set to '13sVZrbGp8ySe1sHYg0d44D9xxVGMI3F'), and 'Client Secret' (represented by a series of asterisks). There are also icons for copy and refresh.

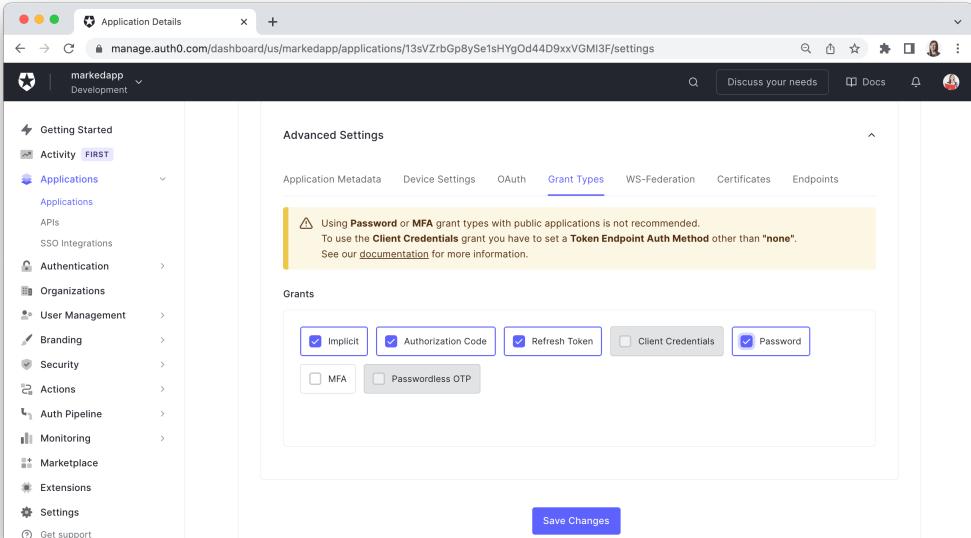
We'll see some important information that we need to send along with a request for an access token, including the Client ID and the Client Secret. Copy the Client ID and Secret values and add them to the `accounts/.env` file now:

`accounts/.env`

```
AUTH0_CLIENT_ID_GRAPHQL=XXXXXXXXXXXXXXXXXXXX  
AUTH0_CLIENT_SECRET_GRAPHQL=XXXXXXXXXXXXXXXXXXXX  
  
# ...
```

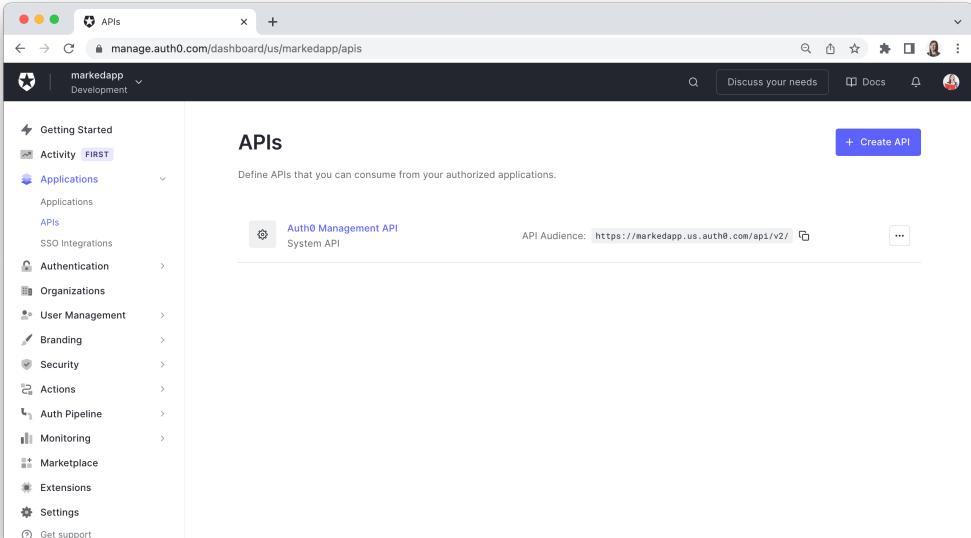
We need to make one small adjustment to the application's settings before we leave this page. Scroll down to the bottom and click on "Advanced Settings" and then select "Grant Types." Check the box for "Password." We can disregard the warning about not using the Password grant type with public applications because for now, we'll only use this application to generate an access token to use with Explorer in the development environment.

Later, we'll also use this application to assist with resetting user account passwords, but that request will be safely contained within the server. Click "Save Changes" now:



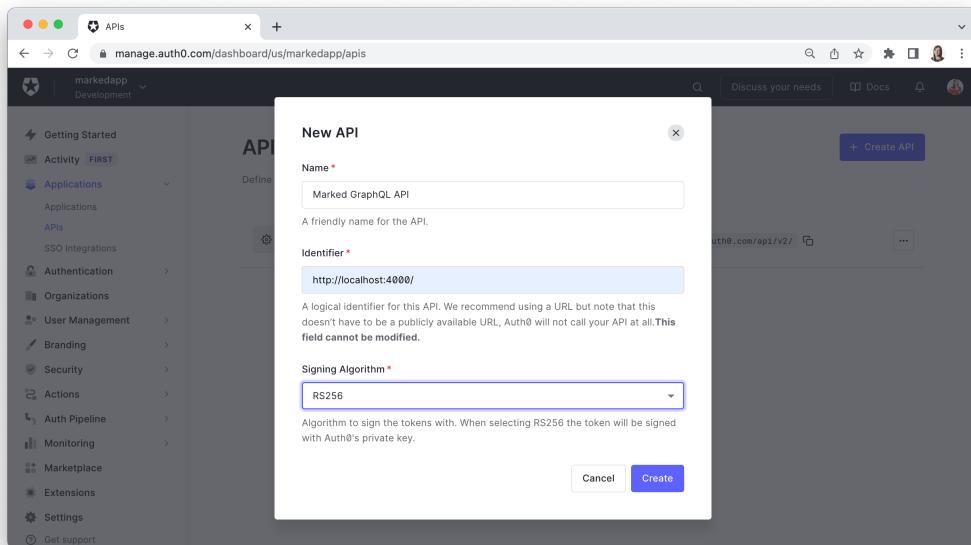
The screenshot shows the Auth0 dashboard for the application 'markedapp'. The left sidebar has 'Activity FIRST' selected under 'Applications'. The main panel is titled 'Advanced Settings' with the 'Grant Types' tab active. A yellow warning box states: '⚠️ Using Password or MFA grant types with public applications is not recommended. To use the Client Credentials grant you have to set a Token Endpoint Auth Method other than "none". See our documentation for more information.' Below this, there are several grant type options: Implicit (checked), Authorization Code (checked), Refresh Token (checked), Client Credentials (unchecked), and Password (checked). There are also two disabled options: MFA and Passwordless OTP. A 'Save Changes' button is at the bottom right.

Next we'll create an Auth0 API, which is an entity that represents some external resource that can accept and respond to protected resource requests. Navigate to the APIs page and click the "Create API" button:

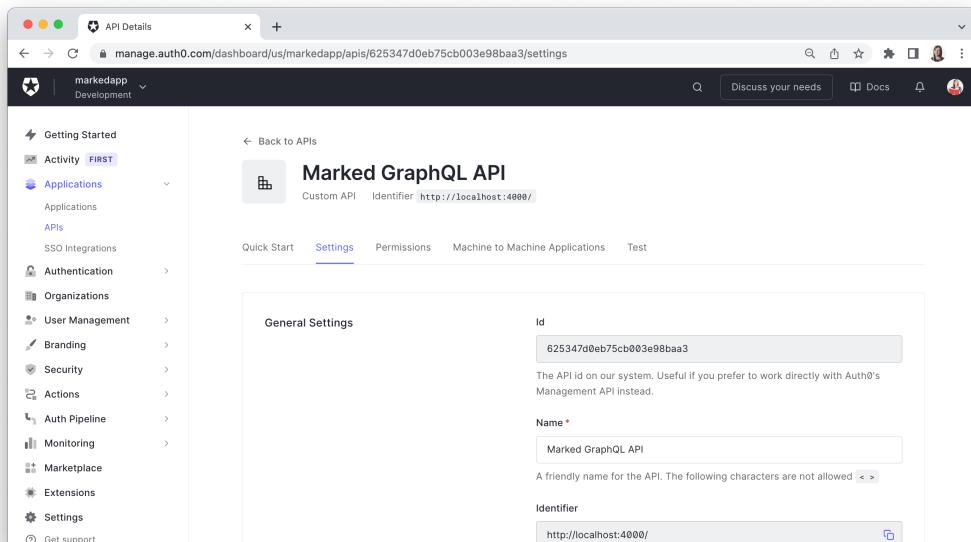


The screenshot shows the Auth0 dashboard on the 'APIs' page for the application 'markedapp'. The left sidebar has 'Activity FIRST' selected under 'Applications'. The main panel is titled 'APIs' and contains the message 'Define APIs that you can consume from your authorized applications.' A 'Create API' button is located in the top right corner. Below it, there is a card for the 'Auth0 Management API' with the sub-type 'System API'. The 'API Audience' field shows the URL: <https://markedapp.us.auth0.com/sapi/v2/>. There is also a '...' button next to the audience field.

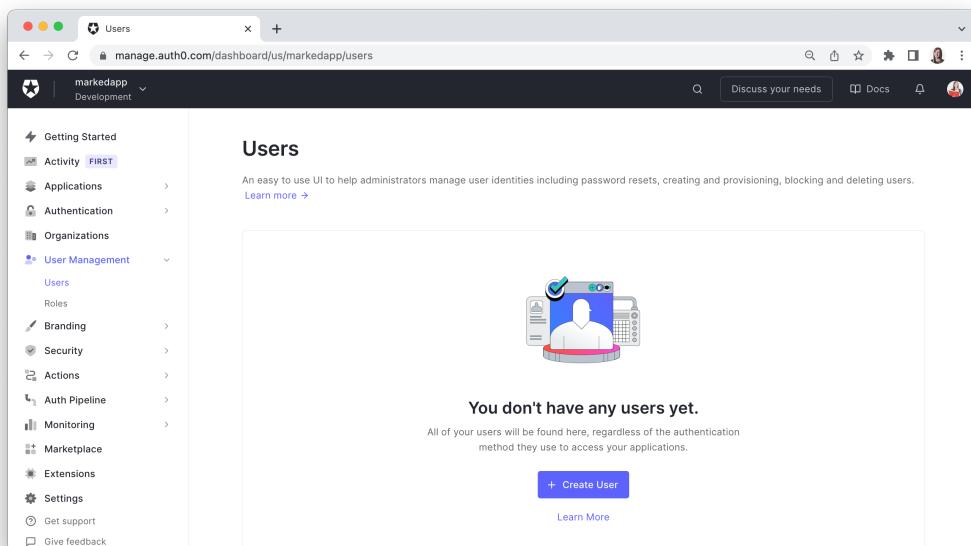
For the identifier, we must use the same AUTH0\_AUDIENCE environment variable value that we previously set:



Once the API is created we will jump to the “Settings” tab:

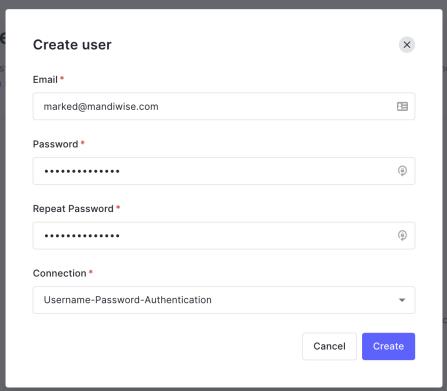


Next, we'll head to the Users page and click the "Create User" button:



The screenshot shows the Auth0 dashboard with the 'Users' page selected. The sidebar on the left has 'User Management' expanded, with 'Users' selected. The main content area displays a message: 'You don't have any users yet.' with a 'Create User' button. There is also a small icon of a person at a computer.

Fill in the form with your email and password:



The screenshot shows the 'Create user' modal window. It contains fields for 'Email' (marked@mandiwise.com), 'Password', 'Repeat Password', and 'Connection' (Username-Password-Authentication). There are 'Cancel' and 'Create' buttons at the bottom.

Ideally, you will use a real email address for this so you can see what the email verification from Auth0 looks like. You can leave the connection as the “Username-Password-Authentication” option. Note that the default password strength requirements for an Auth0 database connection require that passwords are at least eight characters in length with:

- At least one lowercase alphabetical character
- At least one uppercase alphabetical character
- At least one numeric character
- At least one special character

After saving, you’ll be redirected to the User Details for your new user:

The screenshot shows the Auth0 dashboard with the URL `manage.auth0.com/dashboard/us/markedapp/users/YXV0aDAINOM2MjUzNDIiMmZhMDhhZjAwNml4NmI3ZjA`. The left sidebar is titled "markedapp" and includes sections for Getting Started, Activity (FIRST), Applications, Authentication, Organizations, and User Management (which is expanded to show Users, Roles, Branding, Security, Actions, Auth Pipelines, Monitoring, Marketplace, Extensions, Settings, Get support, and Give feedback). The main content area is titled "User Details" for the user `marked@mandiwise.com` (user\_id: auth0|625549b2fa8af806b86b7f8). It shows tabs for Details, Devices, History, Raw JSON, Authorized Applications, Permissions, and Roles. The Details tab displays the following information:

Name	Email	Signed Up
marked@mandiwise.com	marked@mandiwise.com (pending)	April 10th 2022, 3:18:42 PM
<a href="#">Edit</a>	<a href="#">Edit</a>	
Primary Identity Provider	Latest Login	Accounts Associated
Database	Never	None
Browser	-	

A "Multi-Factor Authentication" section is also present at the bottom.

If you used a real email address for the user account, then you will receive an email to verify your account shortly. If you like, you can verify the account and when you return to Auth0 you will see that the user email has changed from “pending” to “verified.”

## Generate a Token for Explorer

We have now completed three of the four required steps to retrieve an access token from Auth0 to use with Explorer. In the final step, we’ll generate an access token for the new user to send with their requests. We won’t be building a real user interface to authenticate users with the Marked app, so we’ll need to take a different approach to log them in and get an access token to include in requests sent from Explorer.

To do that, we'll write a script that we can run from the accounts service. This script will send a request to Auth0 with all of the necessary information about the application, API, and user we just created, and then return the user's access token for us to use in Explorer.

Thinking ahead, we'll place the code that takes care of fetching the access token from Auth0 in a function outside of the script's file because we'll need to reuse it again later in this chapter when we create a mutation to update a user's password. Before we get started with the script, we'll add two new variables to the `.env` file in accounts:

*accounts/.env*

```
AUTH0_AUDIENCE=http://localhost:4000/  
AUTH0_CLIENT_ID_GRAPHQL=XXXXXXXXXXXXXXXXXXXX  
AUTH0_CLIENT_SECRET_GRAPHQL=XXXXXXXXXXXXXXXXXXXX  
AUTH0_DOMAIN=markedapp.us.auth0.com  
  
# ...
```

Again, the `AUTH0_DOMAIN` variable above is for demonstration purposes, so be sure to update it to your unique tenant domain. Next, install the `request` library in accounts:

*accounts/*

```
npm i request@2.88.2
```

Now we'll create a `utils` directory in `accounts/src` and add a `getToken.js` file to it with the following code:

*accounts/src/utils/getToken.js*

```
import util from "util";  
  
import request from "request";  
  
const requestPromise = util.promisify(request);  
  
async function getToken(username, password) {  
  const options = {  
    method: "POST",  
    url: `https://${process.env.AUTH0_DOMAIN}/oauth/token`,  
    headers: {  
      "content-type": "application/x-www-form-urlencoded"  
    },  
    form: {  
      ...  
    }  
  };  
  
  const response = await requestPromise(options);  
  
  if (response.statusCode === 200) {  
    const data = JSON.parse(response.body);  
    return data.access_token;  
  } else {  
    throw new Error(`Error getting token: ${response.statusCode}`);  
  }  
}  
  
module.exports = {  
  getToken  
};
```

```
audience: process.env.AUTH0_AUDIENCE,
client_id: process.env.AUTH0_CLIENT_ID_GRAPHQL,
client_secret: process.env.AUTH0_CLIENT_SECRET_GRAPHQL,
grant_type: "http://auth0.com/oauth/grant-type/password-realm",
password,
realm: "Username-Password-Authentication",
scope: "openid",
username
}
};

const response = await requestPromise(options).catch(error => {
  throw new Error(error);
});
const body = JSON.parse(response.body);
const { access_token } = body;

if (!access_token) {
  throw new Error(
    body.error_description || "Cannot retrieve access token."
  );
}

return access_token;
}

export default getToken;
```

That's a hefty block of code, so let's break it down piece by piece. The `request` module is used to send a POST request to the `/oauth/token` endpoint of Auth0's Authentication API (using your tenant's domain). The `util` library is used to wrap `request` so that it returns a promise. The wrapped request can then be used with `async/await`.

The user's username (which is their email address in our case) and password are passed into the function as arguments and are included as `form` data along with some additional data to authenticate our request with Auth0.

When a response is received we parse it and check if it contains an access token and then return it. If there's no access token in the body of the response, then we throw an error because something went wrong. We'll put the `getToken` function to use by creating an `authenticateUser.js` file in the new `utils` directory with this code in it:

accounts/src/utils/authenticateUser.js

```
import getToken from "./getToken.js";

(async function () {
  const [email, password] = process.argv.slice(2);
  const access_token = await getToken(email, password).catch(error => {
    console.log(error);
  });
  console.log(access_token);
})();
```

In the script above, we slice all of the items after the second index in the `process.argv` array to destructure the `email` and `password` arguments that we will pass to the `getToken` function (the first two arguments in the `process.argv` array are the path to Node.js and the path to the script we are executing).

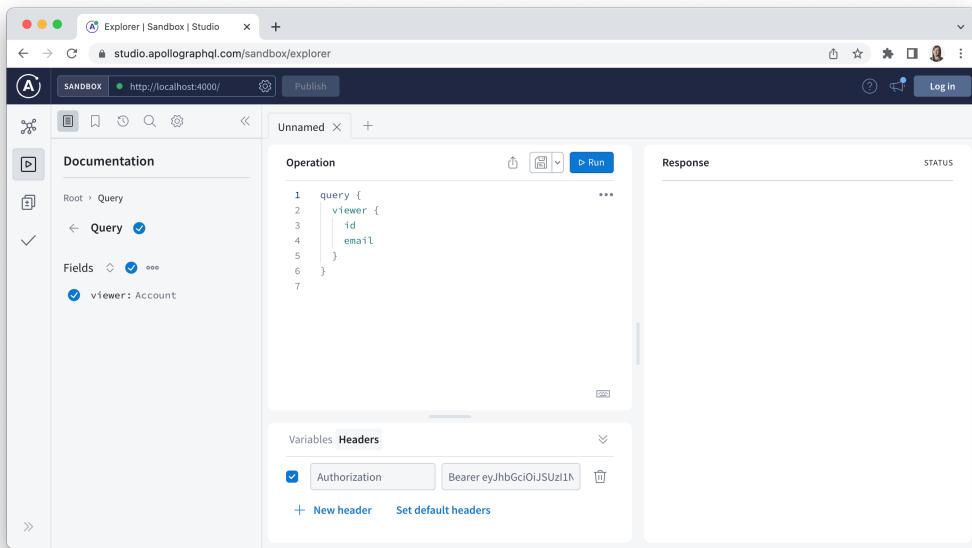
Open a new terminal, `cd` into the `accounts` directory, and run the script:

accounts/

```
node -r dotenv/config src/utils/authenticateUser.js hello@world.com
superHARDpa55!
```

Don't forget to replace the two placeholder arguments above with the real email address and password of the user you previously created in Auth0. If you see an `Unauthorized` error, then double-check and make sure that you added all of the Auth0-related environment variables correctly and that the email and password arguments are correct.

Copy the access token that was logged to the console and head over to Explorer. Open the "Headers" panel below the operation editor and add an `Authorization` header with a value of `Bearer` followed by the token value that was copied from the terminal:



If we now run our `viewer` query again we'll see the decoded payload for the access token JWT logged to the console. It will look roughly like this:

```
{  
  iss: 'https://markedapp.us.auth0.com/',  
  sub: 'auth0|6253805defc16b0068478b45',  
  aud: [  
    'http://localhost:4000/',  
    'https://markedapp.us.auth0.com/userinfo'  
,  
  iat: 1653229222,  
  exp: 1653315622,  
  azp: '13sVZrbGp8ySe1sHYg0d44D9xxVGMI3F',  
  scope: 'openid',  
  gty: 'password'  
}
```

What's particularly useful to us in this token is the `sub` key—this is the Auth0 account ID of the currently logged-in user. We can use it to fetch their specific user data in the next section. Before moving on, we can clean up the `console.log(user);` line from the `viewer` resolver function because we will no longer need it now that we know we can retrieve a decoded token from Auth0.

The JWT issued by Auth0 is only valid for one day, so you will need to rerun this script to generate a new token to use with Explorer every 24 hours.

## The Auth0 Management API

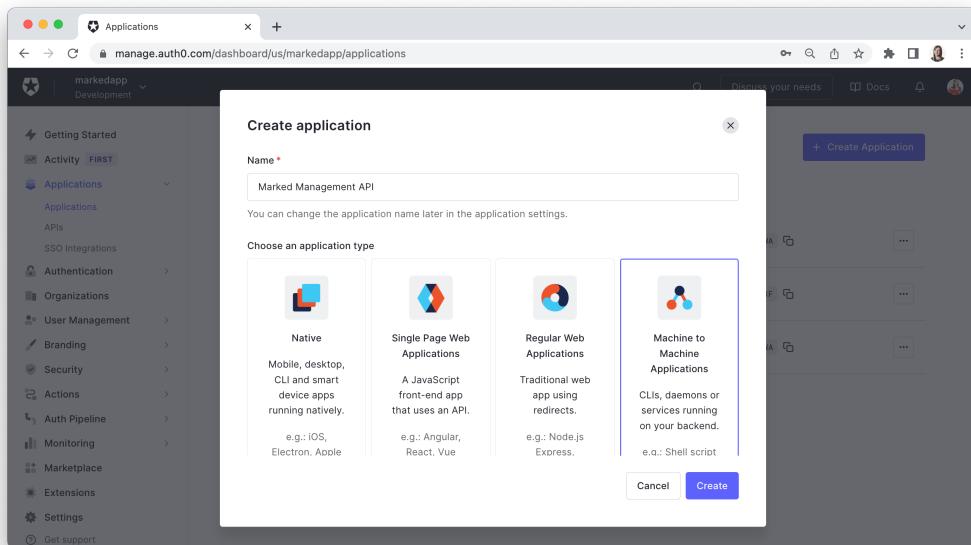
Beyond passing data about a logged-in user into resolvers, we need to perform various operations in Auth0 via these functions as well. Specifically, we need to read and write account data via the GraphQL API rather than just doing these things in the Auth0 dashboard. For that, we will need to make use of [Auth0's Management API](#). The Management API is meant to be used by secure back-end servers and other trusted parties only (so not for front-end clients!) and it allows us to perform a variety of tasks that would otherwise be performed directly through the Auth0 dashboard. That means that this API is very powerful, so it's important to use it with care.

Let's take a quick look at this API's documentation and see what the `users` endpoint can do. Scanning down the left-hand menu of the page we can see exactly the kinds of things we'll need to do to perform CRUD operations on user data in the accounts service's resolvers:

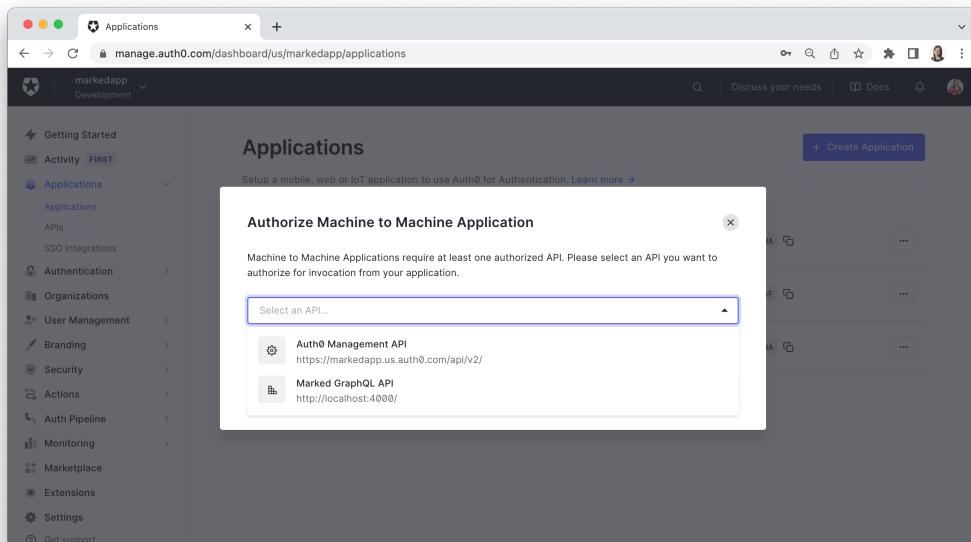
The screenshot shows a web browser window displaying the Auth0 Management API documentation. The URL in the address bar is `auth0.com/docs/api/management/v2#/Users/get_users`. The main content area is titled "List or Search Users". It includes a "GET" button next to the endpoint URL `/api/v2/users`. Below the title, there is a description: "Retrieve details of users. It is possible to:" followed by a bulleted list: "- Specify a search criteria for users", "- Sort the users to be returned", "- Select the fields to be returned", and "- Specify the number of users to retrieve per page and the page index". Further down, there is a note: "The `q` query parameter can be used to get users that match the specified criteria [using query string syntax](#)". There is also a link "Learn more about searching for users.". At the bottom, there is another note: "Read about [best practices](#) when working with the API endpoints for retrieving users." and "Auth0 limits the number of users you can return. If you exceed this threshold, please redefine your search, use the [export job](#), or the [User Import / Export extension](#)". On the left side of the main content, there is a sidebar with a list of management API endpoints: Connections, Custom Domains, Device Credentials, Grants, Hooks, Log Streams, Logs, Organizations, Prompts, Resource Servers, Roles, Rules, Rules Configs, User Blocks, and Users. Under the "Users" section, there are three items: "List or Search Users" (which is currently selected), "Create a User", and "Get a User".

To use the Management API, we need to create a dedicated Auth0 application for this purpose. We'll use the second application to authenticate our Auth0 account in the accounts service so we can access the Management API's endpoints from the field resolvers. To clarify, we're talking about authenticating our overarching Auth0 account here, and not the user account we created for Marked earlier in this chapter.

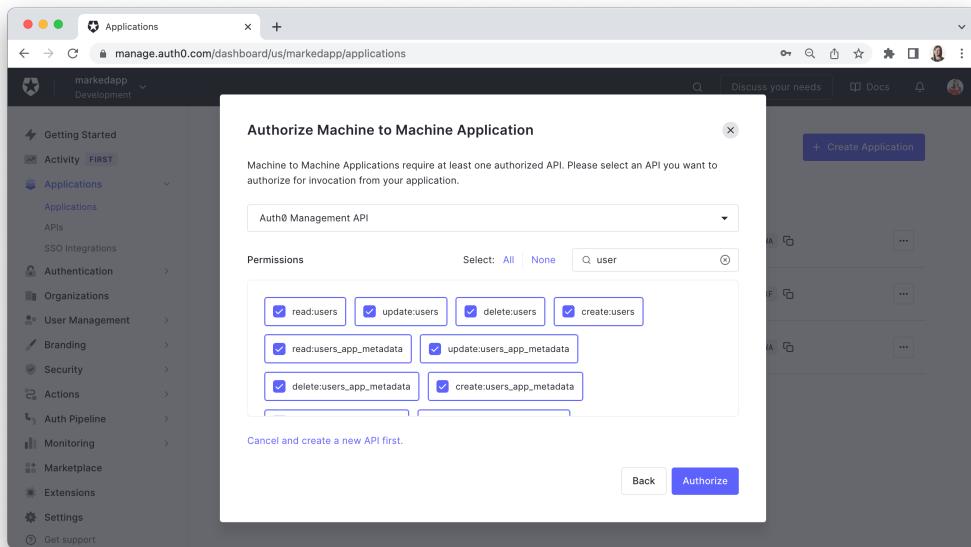
From the Applications page, click the “Create Application” button again. Like before, give it a relevant name but this time choose “Machine to Machine Application” instead:



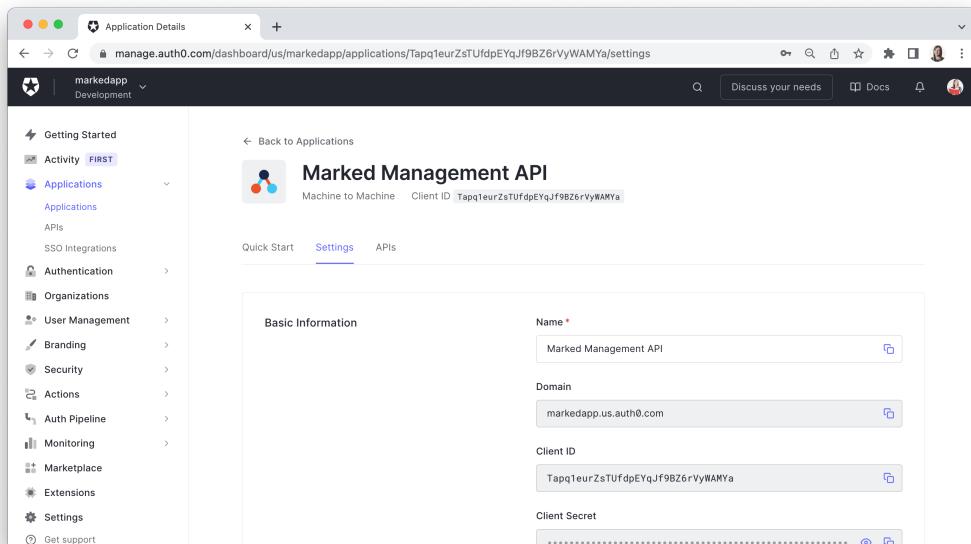
Select the “Auth0 Management API” option to associate with this application:



Now select scopes for the API's access to the application. You can filter for “user” to select all the scopes that apply to reading, creating, updating, and deleting users and their metadata:



After clicking the “Authorize” button we'll navigate to the settings for the application:



Keep the Client ID and Client Secret values handy because we'll need them in the next section to configure an Auth0 Management API client in the accounts service.

## Configure Auth0 with Node.js

To use the Auth0 Management API in the accounts service, we must install a Node.js client library for Auth0 in accounts:

*accounts/*

```
npm i auth0@2.40.0
```

This library will facilitate access to the Auth0 Management API by allowing us to instantiate an authenticated client for this purpose, and then call convenience methods on that client object to perform CRUD operations on Marked user accounts (rather than explicitly making various HTTP requests to the /users endpoint in our code).

Please see [the API documentation for the Auth0 Node.js client](#) for more information on its full capabilities.

We'll need to add two more variables to our `.env` file for the Client ID and Client Secret of the Management API application we just created:

*accounts/.env*

```
# ...  
  
AUTH0_CLIENT_ID_MGMT_API=XXXXXXXXXXXXXXXXXXXX  
AUTH0_CLIENT_SECRET_MGMT_API=XXXXXXXXXXXXXXXXXXXX  
  
# ...
```

Now we'll add a `config` directory inside of `accounts/src` and add an `auth0.js` file to it with the following code:

*accounts/src/config/auth0.js*

```
import { ManagementClient } from "auth0";  
  
const auth0 = new ManagementClient({  
  domain: process.env.AUTH0_DOMAIN,
```

```
    clientId: process.env.AUTH0_CLIENT_ID_MGMT_API,
    clientSecret: process.env.AUTH0_CLIENT_SECRET_MGMT_API
});

export default auth0;
```

With this code in place, we can finally fetch a real user in our `viewer` resolver. First, we'll import the instance of the Auth0 ManagementClient we created in `auth0.js` at the top of the `resolvers.js` file:

`accounts/src/graphql/resolvers.js`

```
import auth0 from "../config/auth0.js";

// ...
```

Now we will call the `getUser` method on the `auth0` object in the `viewer` resolver. We'll set `user.sub` as the value for the `id` property in this method's object argument. For now, let's just log out what we get back from this method before updating what's returned from the `viewer` resolver. You'll see that the `getUser` method returns a promise, so we use the `await` keyword with it and then update the `viewer` resolver to be `async` now:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Query: {
    async viewer(parent, args, { user }) {
      const viewer = await auth0.getUser({ id: user.sub });
      console.log(viewer);
      return accounts[0];
    }
  }
};

export default resolvers;
```

Run the `viewer` query in Explorer again. You will see an object with the following shape logged to the console where the accounts service is running:

```
{  
  created_at: '2022-04-11T01:11:57.126Z',  
  email: 'marked@mandiwise.com',  
  email_verified: false,  
  identities: [  
    {  
      user_id: '6253805defc16b0068478b45',  
      provider: 'auth0',  
      connection: 'Username-Password-Authentication',  
      isSocial: false  
    }  
  ],  
  name: 'marked@mandiwise.com',  
  nickname: 'marked',  
  picture: 'https://s.gravatar.com/avatar/f0242cad4a64bb14314ab1ca791286e7  
?s=480&r=pg&d=https%3A%2F%2Fcdn.auth0.com%2Favatars%2Fma.png',  
  updated_at: '2022-04-10T21:54:40.130Z',  
  user_id: 'auth0|6253805defc16b0068478b45',  
  last_ip: '2001:56a:f868:ab00:2140:fbe5:4d76:5e05',  
  last_login: '2022-04-10T21:54:40.130Z',  
  logins_count: 1  
}
```

Now that we know what user account data Auth0 provides we can build out our `Account` type with a few more fields and create the corresponding resolvers too.

## Add New Account Fields

Our accounts service will only be responsible for managing basic account-related information for users such as their email addresses and the time their accounts were created. All of this data will be stored in and retrieved from Auth0. Additional user metadata will be managed by the profiles service in a later chapter.

Let's update our `Account` type with a new `createdAt` field. We'll also add two new queries to retrieve all user accounts and fetch a single user account by its ID. Lastly, and as a best practice, we will begin adding documentation to our schema with field and type descriptions:

`accounts/src/graphql/schema.graphql`

```
# ...  
  " " "
```

```
An account is a unique Auth0 user.  
"""  
type Account @key(fields: "id") {  
    "The unique ID associated with the account."  
    id: ID!  
    "The date and time the account was created."  
    createdAt: String!  
    "The email associated with the account (must be unique)."  
    email: String!  
}  
  
type Query {  
    "Retrieves a single account by ID."  
    account(id: ID!): Account!  
    "Retrieves a list of accounts."  
    accounts: [Account]  
    "Retrieves the account of the currently logged-in user."  
    viewer: Account  
}
```

Next, we'll need to add the resolvers for the new `createdAt`, `account`, and `accounts` fields. But before doing that, be sure to clean up the `accounts` constant and its dummy data in the `resolvers.js` file because we'll only work with real Auth0 data moving forward. First, we'll add the `account` and `accounts` resolvers and also update the `viewer` resolver to return real data:

`accounts/src/graphql/resolvers.js`

```
import auth0 from "../config/auth0.js";  
  
const resolvers = {  
    // ...  
  
    Query: {  
        account(root, { id }) {  
            return auth0.getUser({ id });  
        },  
        accounts() {  
            return auth0.getUsers();  
        },  
        viewer(root, args, { user }) {  
            if (user?.sub) {  
                return auth0.getUser({ id: user.sub });  
            }  
        }  
    }  
};
```

```
        }
      return null;
    }
};

export default resolvers;
```

Note the changes made to the `viewer` resolver. As per our schema, the `viewer` query can be `null`, so if we check the context for the user and if either the `user` object or its `sub` property is empty, then we return `null` from the resolver. Otherwise, we can use the `user.sub` value to fetch the account data for the currently authenticated user. Returning `null` from this resolver would indicate that the request is being made from an unauthenticated user, which could be helpful for control flow in client applications. Also, note that we have removed the `async` keyword from the `viewer` resolver because we are directly returning the promise returned by `getUser` from it now.

Next, we'll update the field resolvers for `Account`. We have to add resolvers for the `id` and `createdAt` fields to correctly map the data from the Auth0 account object that was retrieved by the parent `account` field:

`accounts/src/graphql/resolvers.js`

```
import auth0 from "../../config/auth0";

const resolvers = {
  Account: {
    __resolveReference(reference) {
      return auth0.getUser({ id: reference.id });
    },
    id(account) {
      return account.user_id;
    },
    createdAt(account) {
      return account.created_at;
    }
  },
  // ...
};

export default resolvers;
```

Note that in the code above we must update the `__resolveReference` method in the `Account` resolvers to fetch a single user account from Auth0 when the `Account` entity is referenced by another subgraph now. It does this using the `account id` property sent in the `reference` object by the referencing subgraph. From Explorer, rerun the `viewer` query with all of its fields:

*GraphQL Query*

```
query Viewer {  
  viewer {  
    id  
    createdAt  
    email  
  }  
}
```

*API Response*

```
{  
  "data": {  
    "viewer": {  
      "id": "auth0|6253805defc16b0068478b45",  
      "createdAt": "2022-04-10T21:18:42.494Z",  
      "email": "marked@mandiwise.com"  
    }  
  }  
}
```

Now try running the `accounts` query:

*GraphQL Query*

```
query Accounts {  
  accounts {  
    id  
    createdAt  
    email  
  }  
}
```

*API Response*

```
{  
  "data": {  
    "accounts": [  
      {  
        "id": "auth0|6253805defc16b0068478b45",  
        "createdAt": "2022-04-10T21:18:42.494Z",  
        "email": "marked@mandiwise.com"  
      }  
    ]  
  }  
}
```

```
"accounts": [
  {
    "id": "auth0|6253805defc16b0068478b45",
    "createdAt": "2022-04-10T21:18:42.494Z",
    "email": "marked@mandiwise.com"
  }
]
```

As expected, we get back an array containing the single user account that we created. Finally, try running the account query, passing the ID for the user account as the `id` argument:

#### *GraphQL Query*

```
query Account($id: ID!) {
  account(id: $id) {
    id
    createdAt
    email
  }
}
```

#### *Query Variables*

```
{
  "id": "auth0|6253805defc16b0068478b45"
}
```

#### *API Response*

```
{
  "data": {
    "account": {
      "id": "auth0|6253805defc16b0068478b45"
      "createdAt": "2022-04-10T21:18:42.494Z",
      "email": "marked@mandiwise.com"
    }
  }
}
```

## Add a `createAccount` Mutation

To finish off this chapter, we'll use the Auth0 API to add the first mutations to the accounts service's schema. First, we'll add a mutation for creating a new user account with a corresponding Input Object type:

`accounts/src/graphql/schema.graphql`

```
# ...

"""
Provides data to create a new account.
"""


```

Next, we'll add the resolver for the `createAccount` mutation to `resolvers.js`. For this resolver, we'll use the `createUser` method from the Auth0 client library. This method expects us to pass in an object as an argument containing a connection type (in our case, `Username-Password-Authentication`) and an email and password for the new user:

`accounts/src/graphql/resolvers.js`

```
import auth0 from "../../config/auth0";

const resolvers = {
    // ...

    Query: {
        // ...
    },
}
```

```
Mutation: {
  createAccount(parent, { input: { email, password } }) {
    return auth0.createUser({
      connection: "Username-Password-Authentication",
      email,
      password
    });
  }
};

export default resolvers;
```

Back in Explorer, the new mutation should be available now. Add the `createAccount` mutation to the operation editor and run it to confirm that a new user is created:

#### *GraphQL Mutation*

```
mutation CreateAccount($input: CreateAccountInput!) {
  createAccount(input: $input) {
    id
    createdAt
    email
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "email": "mark@markedmail.com",
    "password": "superHARDpa55!"
  }
}
```

#### *API Response*

```
{
  "data": {
    "createAccount": {
      "id": "auth0|62536171efc16b006847886a",
      "createdAt": "2022-04-10T23:00:01.651Z",
      "email": "mark@markedmail.com",
      "password": "superHARDpa55!",
      "connection": "Username-Password-Authentication"
    }
  }
}
```

```
        "email": "mark@markedmail.com"
    }
}
```

## Add Mutations to Update Accounts

Now that we have a couple of user accounts to work with in Auth0, let's add mutations that update the emails or passwords associated with them. If we take a look at the documentation for Auth0's `updateUser` method, then we can see that it expects us to provide `params` for selecting which account to update (specifically, the user's ID) as well as the `data` that we wish to update in that user's account.

As far as account management goes, users may need to update their password or their email. Typically, a user wouldn't update both of these values at the same time so, as a schema design best practice, we will create two finer-grained mutations for these account actions rather than one all-encompassing update mutation. Creating purpose-built fields that are tailored for client use cases is one of the main advantages of using GraphQL to build an API!

We'll begin by adding two Input Object types to support these mutations:

`accounts/src/graphql/schema.graphql`

```
# ...

"""
Provides data to update an existing account's email.
"""

 UpdateAccountEmailInput {
  "The unique ID associated with the account."
  id: ID!
  "The updated account email."
  email: String!
}

"""
Provides data to update an existing account's password. A current password
and a new password are required to update a password.
"""

 UpdateAccountPasswordInput {
  "The unique ID associated with the account."
  id: ID!
```

```
"The updated account password."
newPassword: String!
"The existing account password."
password: String!
}

# ...
```

Now that our update-related Input Object types are ready we can create the new fields on the root `Mutation` type:

`accounts/src/graphql/schema.graphql`

```
# ...

type Mutation {
  "Creates a new account."
  createAccount(input: CreateAccountInput!): Account!
  "Updates an account's email."
  updateAccountEmail(input: UpdateAccountEmailInput!): Account!
  "Updates an account's password."
  updateAccountPassword(input: UpdateAccountPasswordInput!): Account!
}
```

Over in `resolvers.js`, we'll import the access token-fetching function from `getToken.js` and also import `UserInputError` from `apollo-server`:

`accounts/src/graphql/resolvers.js`

```
import { UserInputError } from "apollo-server";

import auth0 from "../config/auth0.js";
import getToken from "../utils/getToken.js";

// ...
```

We'll use `getToken` here due to the sensitive nature of updating an account's password. Rather than blindly overwriting the old password value with a new one, we want to check to make sure the person submitting the request knows the account's existing password. To do this, we'll verify that we can fetch an access token for the user account first using their old password, and then proceed with updating the password to the new one.

First, we'll implement the resolver for the `updateAccountEmail` mutation using the `updateUser` method on the `auth0` object:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Mutation: {
    createAccount(root, { input: { email, password } }) {
      // ...
    },
    updateAccountEmail(root, { input: { id, email } }) {
      return auth0.updateUser({ id }, { email });
    }
  }
};

export default resolvers;
```

Next, we'll add the `updateAccountPassword` mutation, which will be a little more involved than the last one. Before updating the password we will fetch the user's account data from Auth0 so we can get their email address. Once we have that we can test if their email and current password combination can be used to authenticate the user successfully. If one of the values is incorrect, we'll throw a `UserInputError`, otherwise, we'll proceed with updating the password:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Mutation: {
    // ...
    updateAccountEmail(root, { input: { id, email } }) {
      return auth0.updateUser({ id }, { email });
    },
    async updateAccountPassword(root, { input: { id, newPassword, password } }) {
      const user = await auth0.getUser({ id });
```

```
try {
  await getToken(user.email, password);
} catch {
  throw new UserInputError("Email or existing password is
    incorrect.");
}

return auth0.updateUser({ id }, { password: newPassword });
}
};

export default resolvers;
```

To test out our changes, run a mutation in GraphQL to update the email for an existing account:

#### *GraphQL Mutation*

```
mutation UpdateAccountEmail($input: UpdateAccountEmailInput!) {
  updateAccountEmail(input: $input) {
    id
    email
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "id": "auth0|62536171efc16b006847886a",
    "email": "marcus@markedmail.com"
  }
}
```

#### *API Response*

```
{
  "data": {
    "updateAccountEmail": {
      "id": "auth0|62536171efc16b006847886a",
      "email": "marcus@markedmail.com"
    }
  }
}
```

```
        }
    }
}
```

Now try the `updateAccountPassword` mutation. We can't query a user's account password as a field on the `Account` type (and thankfully so, because this would be very dangerous!), so request their `id` field after the mutation instead to ensure the write operation was a success:

#### *GraphQL Mutation*

```
mutation UpdateAccountPassword($input: UpdateAccountPasswordInput!) {
  updateAccountPassword(input: $input) {
    id
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "id": "auth0|625349b2fa08af006b86b7f0",
    "newPassword": "superHARDpa56!",
    "password": "superHARDpa55!"
  }
}
```

#### *API Response*

```
{
  "data": {
    "updateAccountPassword": {
      "id": "auth0|62536171efc16b006847886a"
    }
  }
}
```

## Add a `deleteAccount` Mutation

Last but not least, we'll take care of the “D” in CRUD for account operations by adding a `deleteAccount` mutation. This mutation will be the easiest to implement yet. If we take a look at

the Auth0 API docs, we'll see that its `deleteUser` method expects us to pass in a `params` object containing the user's ID. First, we'll add the mutation to the schema:

`accounts/src/graphql/schema.graphql`

```
# ...

type Mutation {
  # ...
  "Deletes an account."
  deleteAccount(id: ID!): Boolean!
  # ...
}
```

Note that this mutation will return `true` or `false` to indicate whether the operation was a success. Now we can add its resolver:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Mutation: {
    // ...
    async deleteAccount(root, { id }) {
      try {
        await auth0.deleteUser({ id });
        return true;
      } catch {
        return false;
      }
    },
    // ...
  }
};

export default resolvers;
```

Note that we must mark this resolver as `async` because we have to wait for the promise to resolve as we wait for a response for the Auth0 API before returning the boolean value. Heading back over to Explorer, we can run our `deleteAccount` mutation on the user we've been working with:

*GraphQL Mutation*

```
mutation DeleteAccount($id: ID!) {
  deleteAccount(id: $id)
}
```

*Mutation Variables*

```
{
  "id": "auth0|62536171efc16b006847886a"
}
```

*API Response*

```
{
  "data": {
    "deleteAccount": true
  }
}
```

Try running the `accounts` query now to double-check that the account was deleted.

## Summary

We accomplished a great deal in this chapter—from signing up for and configuring Auth0, to adding Express middleware to verify JWTs, and finally adding essential queries and mutations that allow us to read, add, update, and remove Auth0-based user accounts from Explorer using our API. With basic authentication and user account management functionality in place, we’re ready to begin building out the API’s authorization layer in the next chapter. But first, we will do some light refactoring in the `accounts` service and also create a shared library for types and directives that will be used across multiple subgraph schemas.

## Chapter 3

# Apollo Data Sources, Custom Scalars, and Custom Directives

In this chapter, we will:

- Refactor data-fetching logic from resolvers into an Apollo data source
- Create a library to facilitate sharing of types and directives across subgraph schemas
- Add a custom `DateTime` Scalar type
- Add custom type system directives to manage field-level authorization

## Manage Data-Fetching Logic in an Apollo Data Source

At the moment, the `resolvers.js` file for the accounts service is a bit messy. This mess has accumulated here because all of the logic for making requests to the Auth0 Management API is handled directly in the resolver code blocks. If this service's schema continues to grow in the future, then this file may become unwieldy.

A better practice would be to encapsulate all of our data-fetching logic for the service elsewhere. Apollo Server provides us with a built-in way to do this using a *data source*. An Apollo data source can also provide mechanisms for caching, de-duplication, and error handling of requests made to some source of data like a REST API. As a bonus, we'll see that an Apollo data source will help make the data-fetching code in the resolvers DRYer and make it easier for us to integrate the DataLoader library in Chapter 7.

To create a data source, we will extend Apollo Server's `DataSource` class and add methods inside of it that fetch data from the Auth0 Management API. Once it's created, we'll add it to Apollo Server's `context` object so that the data source methods are available to all resolvers.

Adding an Apollo data source will require some refactoring in the `resolvers.js` file now (for future services, we'll use data sources from the get-go to avoid this). We'll start by installing the `apollo-datasource` package in `accounts`:

accounts/

```
npm i apollo-datasource@3.3.1
```

Next, we'll create a subdirectory called `dataSources` in `accounts/src/graphql` and add an `AccountsDataSource.js` file to it with the following initial code:

`accounts/src/graphql/dataSources/AccountsDataSource.js`

```
import { DataSource } from "apollo-datasource";

class AccountsDataSource extends DataSource {
  constructor({ auth0 }) {
    super();
    this.auth0 = auth0;
  }

  // Data-fetching methods will go here...
}

export default AccountsDataSource;
```

We pass `auth0` into the constructor and set it as a property because when we instantiate our data source later we'll pass in our Auth0 configuration so we have access to the Management API client. This change means that wherever we called an `auth0` method in a resolver previously we'll need to update it to `this.auth0` now as we refactor that code and add it to the `AccountsDataSource`.

Now we'll migrate the data-fetching code in the resolvers into some methods for this class. We'll begin by adding a `createAccount` method that has `email` and `password` parameters to support user account creation in Auth0:

`accounts/src/graphql/dataSources/AccountsDataSource.js`

```
import { DataSource } from "apollo-datasource";

class AccountsDataSource extends DataSource {
  // ...

  createAccount(email, password) {
    return this.auth0.createUser({
      connection: "Username-Password-Authentication",
      email,
      password
    })
  }
}
```

```
    });
  }
}

export default AccountsDataSource;
```

Again, note that we call Auth0's `createUser` method on `this.auth0` now. Next, we'll add methods to support email and password updates. To do that, we must move the `UserInputError` and `getToken` imports from the `resolvers.js` file to `AccountsDataSource.js` because we'll do the token fetching and error handling here instead:

`accounts/src/graphql/dataSources/AccountsDataSource.js`

```
import { DataSource } from "apollo-datasource";
import { UserInputError } from "apollo-server";

import getToken from "../../utils/getToken.js";

// ...
```

Now we can add `updateAccountEmail` and `updateAccountPassword` methods:

`accounts/src/graphql/dataSources/AccountsDataSource.js`

```
// ...

class AccountsDataSource extends DataSource {
  // ...

  updateAccountEmail(id, email) {
    return this.auth0.updateUser({ id }, { email });
  }

  async updateAccountPassword(id, newPassword, password) {
    const user = await this.auth0.getUser({ id });

    try {
      await getToken(user.email, password);
    } catch {
      throw new UserInputError("Email or existing password is incorrect.");
    }
  }
}
```

```
        return this.auth0.updateUser({ id }, { password: newPassword });
    }
}

export default AccountsDataSource;
```

Lastly, we'll create a `deleteAccount` method, giving it an `id` parameter

*accounts/src/graphql/dataSources/AccountsDataSource.js*

```
// ...

class AccountsDataSource extends DataSource {
// ...

    async deleteAccount(id) {
        try {
            await this.auth0.deleteUser({ id });
            return true;
        } catch {
            return false;
        }
    }
}

export default AccountsDataSource;
```

Apart from the write operations, we also need methods to read data from Auth0 in this data source. The first method will be called `getAccountById`:

*accounts/src/graphql/dataSources/AccountsDataSource.js*

```
// ...

class AccountsDataSource extends DataSource {
// ...

    getAccountById(id) {
        return this.auth0.getUser({ id });
    }
}

// ...
```

```
}

export default AccountsDataSource;
```

This method will work in both the `account` and `viewer` resolvers because these two resolvers fetch the same kind of data and in the same way, but with ID arguments originating from different sources (the `account` query has a user-supplied `id` argument while `viewer` uses the decoded JWT to get the account ID).

Lastly, we'll create the accompanying `getAccounts` method:

*accounts/src/graphql/dataSources/AccountsDataSource.js*

```
// ...

class AccountsDataSource extends DataSource {
  // ...

  getAccounts() {
    return this.auth0.getUsers();
  }

  // ...
}

export default AccountsDataSource;
```

Now that we've created our data source we need to add it to the accounts service's Apollo Server context. First, we'll import the `AccountsDataSource` class into the `index.js` file. Because we pass the `auth0` client into the `AccountsDataSource` constructor, we need to import it here too:

*accounts/src/index.js*

```
// ...

import AccountsDataSource from
  './graphql/dataSources/AccountsDataSource.js';
import auth0 from './config/auth0.js';
import resolvers from './resolvers.js';

// ...
```

Next, we can instantiate a new `AccountsDataSource` object and pass in the `auth0` client to it. As mentioned previously, the data source will be available in the context of the resolvers. Rather than adding it imperatively to the context, we set a dedicated `dataSources` option in the `ApolloServer` constructor. This method returns an object containing all of the data sources we want to access from our resolver functions:

`accounts/src/index.js`

```
// ...

const server = new ApolloServer({
  schema: buildSubgraphSchema({ typeDefs, resolvers }),
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user };
},
  dataSources: () => {
    return {
      accountsAPI: new AccountsDataSource({ auth0 })
    };
}
));

// ...
```

With this code in place, we can access the `AccountsDataSource` methods in the resolvers. To do this, we destructure the `dataSources` object from the resolver's `context` parameter. From there, the methods we created can be called from `accountsAPI` object nested inside it. We'll update the field resolvers for the root `Query` type first:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Query: {
    account(root, { id }, { dataSources }) {
      return dataSources.accountsAPI.getAccountById(id);
    },
    accounts(root, args, { dataSources }) {
      return dataSources.accountsAPI.getAccounts();
    }
  }
};

// ...
```

```
  },
  viewer(root, args, { dataSources, user }) {
    if (user?.sub) {
      return dataSources.accountsAPI.getAccountById(user.sub);
    }
    return null;
  },
},
// ...
};

export default resolvers;
```

We can even use the `getAccountById` method in the `__resolveReference` resolver for the `Account` entity because the context object is available as its the second parameter:

`accounts/src/graphql/resolvers.js`

```
// ...

const resolvers = {
// ...

  Account: {
    __resolveReference(reference, { dataSources }) {
      return dataSources.accountsAPI.getAccountById(reference.id);
    },
    // ...
  }

// ...
};

export default resolvers;
```

Lastly, we'll update the `Mutation` resolvers:

`accounts/src/graphql/resolvers.js`

```
// ...
```

```
const resolvers = {
  // ...

  Mutation: {
    createAccount(root, { input: { email, password } }, { dataSources }) {
      return dataSources.accountsAPI.createAccount(email, password);
    },
    deleteAccount(root, { id }, { dataSources }) {
      return dataSources.accountsAPI.deleteAccount(id);
    },
    updateAccountEmail(root, { input: { id, email } }, { dataSources }) {
      return dataSources.accountsAPI.updateAccountEmail(id, email);
    },
    updateAccountPassword(
      root,
      { input: { id, newPassword, password } },
      { dataSources }
    ) {
      return dataSources.accountsAPI.updateAccountPassword(
        id,
        newPassword,
        password
      );
    }
  }
};

export default resolvers;
```

Take note that the `deleteAccount` and `updateAccountPassword` resolvers no longer need to be `async` because they will just return the promise returned by their respective data source methods. Additionally, because our resolvers are now fully refactored to use the data source, we can delete the `auth0` import at the top of `resolvers.js` too as we will no longer need direct access to it here. Before moving on, you'll want to retest all of the accounts service's queries and mutations in Explorer to make sure they still work as they did before refactoring.

## Add a Custom DateTime Scalar Type

GraphQL has `Int`, `Float`, `String`, `Boolean`, and `ID` Scalar types by default, but sometimes these built-in types aren't quite enough! When warranted, we can add a custom Scalar type with a workflow that we already know well—in other words, we'll define a type in the schema and then write a resolver so that the type will be handled as desired during GraphQL execution.

We're going to add a custom `DateTime` Scalar type to our schema and use it as an output type for the `createdAt` field. While we can be reasonably confident that Auth0 will provide us with date values that are in a valid ISO 8601 format, the `DateTime` Scalar will provide an extra measure of certainty and transparency for the consumers of our API that they will receive a date string in this precise format. This Scalar type will also be a convenient way to enforce consistency for all date-related fields across the different subgraphs we create.

Federation 2 offers more flexibility than the Federation 1 specification did when it comes to sharing non-entity types across subgraphs. We'll explore some of these features in more depth in Chapter 5, but for now, all that we need to know is that multiple subgraphs can define and use the same custom Scalar types. But it's up to the teams that own the subgraph schemas to ensure that they implement consistent serialization and parsing logic for the Scalar type to ensure predictability for the clients that send requests to the API.

To that end, we're going to create a library to contain shared custom type definitions so that we can use them consistently across subgraph schemas without having to redefine them explicitly in each one. For brevity's sake, we will keep these definitions in a sibling directory called `shared` alongside the `accounts` and `gateway` directories in our project. However, we could imagine that this library would exist in a separate repository and would be installed as a dependency in any Node.js-based subgraphs.

Create a subdirectory called `shared` in the root project directory and add a `package.json` file to it:

```
shared/
```

```
npm init --yes
```

Next, we'll opt into using `import/export` syntax and also set the entry point for this module:

```
shared/package.json
```

```
{
  // ...
  "type": "module",
  "main": "src/index.js",
  // ...
}
```

We'll need to install a few familiar packages in `shared` to support the `DateTime` Scalar, and we will also need the `validator` package to assist with date string validation:

```
npm i apollo-server@3.7.0 graphql@16.5.0 validator@13.7.0
```

Next, we'll add a `src` subdirectory inside of `shared` and then a `scalars` subdirectory inside of `src` with a `DateTimeType.js` file in it that contains the following initial code:

`shared/src/scalars/DateTimeType.js`

```
import { ApolloError } from "apollo-server";
import { GraphQLScalarType } from "graphql";
import validator from "validator";

const DateTimeType = new GraphQLScalarType({
  name: "DateTime",
  description: "An ISO 8601-encoded UTC date string."
  // Date string validation logic will go here...
});

export default DateTimeType;
```

The value of `DateTimeResolver` must be a newly instantiated `GraphQLScalarType` object. This constructor expects us to pass in an object with a `name` and a `description` for the custom Scalar type. In addition to that, we have to create three methods that set the rules for the expected input and output values:

- `parseValue` ensures the value sent *from* the client is a valid date string
- `serialize` ensures the value to be sent *to* the client is a valid date string
- `parseLiteral` ensures the GraphQL abstract syntax tree (AST) value is a valid date string

To do this validation, our final `DateTimeType` will look like this:

`shared/src/scalars/DateTimeType.js`

```
// ...

const DateTimeType = new GraphQLScalarType({
  name: "DateTime",
  description: "An ISO 8601-encoded UTC date string.",
  parseValue: value => {
    if (validator.isISO8601(value)) {
      return value;
    }
    throw new ApolloError("DateTime must be a valid ISO 8601 Date string");
  },
  serialize: value => {
    if (typeof value !== "string") {
      value = value.toISOString();
```

```
}

if (validator.isISO8601(value)) {
  return value;
}
throw new ApolloError("DateTime must be a valid ISO 8601 Date string");
},
parseLiteral: ast => {
  if (validator.isISO8601(ast.value)) {
    return ast.value;
  }
  throw new ApolloError("DateTime must be a valid ISO 8601 Date string");
}
);

export default DateTimeType;
```

When we serialize the data to be sent to the client we do an extra check to see if the value is a string first. We do this check here because later on we'll see that dates are stored as Date objects in MongoDB, and we'll need to convert those objects to ISO 8601 date strings before we run a function to confirm that they are in the correct format.

We have one last thing to do in `shared`, which is to configure the main entry point for this library, which will be an `index.js` file in `src`:

`shared/src/index.js`

```
import DateTimeType from "./scalars/DateTimeType.js";

export { DateTimeType };
```

Now we're ready to define the `DateTime` Scalar type in the accounts service's schema and use it as the output type for the `createdAt` field:

`accounts/src/graphql/schema.graphql`

```
# ...

scalar DateTime

"""
An account is a unique Auth0 user.
"""
```

```
type Account @key(fields: "id") {
  "The unique ID associated with the account."
  id: ID!
  "The date and time the account was created."
  createdAt: DateTime!
  "The email associated with the account (must be unique)."
  email: String!
}

# ...
```

Like most things in a GraphQL schema, the custom Scalar type needs to be represented in this subgraph's map of resolvers. It's worth noting again that the code in the shared directory would likely live in a distinct package that could be installed in any Node.js-based subgraph. For instructional purposes, we will keep things simple and just relatively import the `DateTimeType` from shared into the accounts service's `resolvers.js` file:

`accounts/src/graphql/resolvers.js`

```
import { DateTimeType } from "../../shared/src/index.js";

const resolvers = {
  DateTime: DateTimeType,

  // ...
};

export default resolvers;
```

Before testing the new output type for the `createdAt` field, we'll preemptively deal with an error that we will encounter due to how our shared module is configured locally. Both the `accounts` and `shared` directories have the `graphql.js` package installed. If we leave things as they are, we will eventually encounter an error like this one while running the `accounts` service:

```
Error: Cannot use GraphQLObjectType "SomeTypeName" from another module or realm.
```

Ensure that there is only one instance of ``graphql`` in the `node_modules` directory. If different versions of ``graphql`` are the dependencies of other relied-on modules, use `"resolutions"` to ensure only one version is installed.

If the `shared` library was truly an independent module, then the `graphql` package would be deduped when installed. However, we have loaded it relatively here instead so from the accounts service's perspective we are trying to load two versions of this package, resulting in the above error. The solution to this problem for our development environment is to use `npm link` to create two symbolic links. The first step is to `cd` into `shared/node_modules/graphql` and run the following command:

```
shared/node_modules/graphql
```

```
npm link
```

Doing this creates a symlink in the global `node_modules` directory that links to the package where the `npm link` command was executed. In other words, we can use the local `graphql` module installed in `shared/node_modules` as a global node module elsewhere, such as in the accounts service.

You can confirm that the package was successfully linked by running this command:

```
npm ls -g --depth=0 --link=true
```

Next, we need to create a symlink in the `accounts` directory that points to the previously linked `graphql` module:

```
accounts/
```

```
npm link graphql
```

As you continue to work through this book, you may find that you need to relink `graphql` after installing other dependencies in the service directories. You will know you need to do this whenever you see `Error: Cannot use GraphQLObjectType "SomeTypeName" from another module or realm.` in the terminal when you try to restart the service. Just run `npm link graphql` again in the service directory and the error will be fixed.

Additionally, if you want to intentionally remove the symlinks you just created, you can run `npm unlink --no-save graphql` in the `accounts` directory first, and then `npm unlink` in `shared/node_modules/graphql`. The order of those commands matters!

If we return to Explorer now, then we should see the new `DateTime` type as the output type for the `createdAt` field in the API documentation and the accounts-related query operations should work just as they did before.

## Approaches to Authorization in GraphQL

The next addition to our library of shared types will be custom type system directives that apply field-level authorization to a selection of fields in the accounts service’s schema. Before we do, let’s pause and consider why we need to do this and what other options we could explore for adding authorization to a GraphQL API.

At the moment, all of the mutations we have written perform user administration tasks in Auth0 and they are fully exposed to anyone who wants to send an operation to our API. Similarly, all of the accounts data is publicly queryable through the API, including the user’s email addresses. In fact, the only field that uses the access token of an authenticated user is the `viewer` field on the root `Query` type and that’s just so we can see information about the currently logged-in user (based on the `sub` claim in the token). No errors will be thrown by any of the queries or mutations if the user making the request isn’t authorized to do so.

But what does being “authorized to do” mean in the Marked application? For the accounts service, it will mean the following:

- Only authenticated users can query data about other accounts
- Only the owner of a given account can update their data or delete the account altogether

So that means for the `account` and `accounts` queries we’ll need to check for a valid access token before returning any data. Similarly, for the `updateAccountEmail`, `updateAccountPassword`, and `deleteAccount` mutations we’ll need to make sure that the ID corresponding to the account to be updated matches the ID of the authenticated user.

There are several options we can choose from to apply these authorization rules on a per-field basis. Specifically, we could:

- Write code in every resolver that validates authorization logic for the field before it proceeds with the operation
- Use resolver middleware to abstract the authorization logic into a separate layer
- Add custom type system directives to validate authorization logic on a query-by-query and mutation-by-mutation basis

The first option would likely be a bit tedious so we’ll avoid handling authorization on a per-resolver basis wherever possible (there are a couple of notable exceptions to this that will be called out later in the book). The second option is typically much DRYer than the first and there are packages such as [GraphQL Shield](#) and [AuthZ](#) that can support this approach. Ultimately, we’ll choose the third option so that we can explore an advanced implementation of custom type system directives at the subgraph level of a federated GraphQL API.

As we move forward through the remainder of the chapter, keep in mind that authorization directives are challenging to implement and there are serious, real-world consequences for doing so incorrectly. Even though our custom directives will end up being quite complex, writing a full-featured series of auth-related directives is outside the scope of this book. However, the code that we write should give you insight into how this task can be approached, provide you with a

foundation to build on in the future, and give you a sense of the relative pros and cons of using directives for authorization.

## Custom Directives with Apollo Federation

Before we dive into writing the code for the authorization directives, we need to understand how custom directives work in Apollo Federation, as well as in GraphQL in general. GraphQL has two different kinds of directives—there are *type system* directives (also known as *schema* directives) and *executable* directives (also known as *query* directives).

Type system directives are used to annotate different elements of an SDL-based schema such as types, fields, and Enum values. Executable directives are used to annotate parts of an operation document such as fragments, field selections, and even an entire query, mutation, or subscription. GraphQL’s built-in `@deprecated` directive is an example of a type system directive, while the `@skip` and `@include` directives are examples of executable directives.

At the time of writing, Apollo Federation provides different kinds of support for custom type system and executable directives. Executable directives can be intercepted by an Apollo Gateway even though Apollo Server itself does not support them. We won’t be adding any executable directives to any of the subgraphs that we create for this API, but it’s good to know that the option is available.

Type system directives, on the other hand, can be applied throughout a subgraph schema just as they would be in a non-federated schema, but a key difference here is that these directives do not roll up to the gateway. In other words, the subgraph will be aware of where the type system directives are located throughout its schema but they will not be present in the supergraph schema because they will be discarded during composition. For most use cases, that will be fine because the subgraph can still apply whatever logic needs to be applied based on the presence of the directive during field resolution.

Lastly, as with any shared definition, when using the same custom type system directive across subgraphs it’s typically a good idea to handle them using consistent logic or ambiguity may result on the client side. Additionally, if multiple subgraphs can resolve the same field for a given type, then each subgraph should also apply the same custom directives with the same handling logic to maintain consistency regardless of where field resolution happens.

As we can see, custom directives should be approached with a certain amount of care and foresight when building a federated graph. But if there is one key takeaway from this discussion, it’s to be consistent and provide predictable results to clients. Importing the authorization directives from a shared library will help satisfy these criteria for the Marked application. However, do keep in mind that if any non-Node.js subgraphs were later composed into the supergraph, then this logic would need to be replicated in that language to use the directives in those subgraphs as well.

## Handling Authorization with Type System Directives

Before we dive into the code, it should be noted that there are many different approaches we could take to implement authorization directives for a GraphQL API depending on the specific requirements. Our authorization system will be relatively simple in that we won't need to implement roles or permissions for users. Instead, there are two key things that we may need to verify before a field resolver executes based on a request from an end user:

- *Is the user authenticated?* If so, then they will have access to certain `Query` and `Mutation` fields that should be unavailable to unauthenticated users.
- *Is the user that sent the operation the owner of the resource that they are trying to access or update?* If they are not, then they will be barred from viewing or making a change to that resources.

In Chapter 9, we'll add a third consideration when checking custom scopes that will be added to certain JWTs to permit access to various fields for back-end services only. These scopes will not apply to end users of the Marked app so they will not be the focus of this chapter.

We could potentially handle the two authorization criteria listed above in a single `@auth` directive with a few nullable arguments to handle different authorization scenarios. However, as with fields, including multiple nullable arguments to a generalized directive to accommodate different scenarios can create ambiguity when applying that directive throughout a schema. It also forces runtime code to take care of error-handling logic for non-sensical argument combinations, so it would be a better idea to create more than one directive to serve these specific purposes.

With that in mind, we will create an `@private` directive to check if an end user is authenticated and an `@owner` directive to verify that they own a resource before accessing it. As previously mentioned, we will build on the foundation we create in the steps that follow with an additional `@scope` directive in Chapter 9.

The [GraphQL Tools](#) library provides a set of utilities that will support the implementation of our authorization directives, so we'll install that in `shared` first:

`shared/`

```
npm i @graphql-tools/utils@8.6.7
```

We'll define the `@private` directive first because it's more straightforward than `@owner`. We'll create a `directives` directory inside of `shared/src` first and then add an `authDirectives.graphql` file to it with the following code:

`shared/src/directives/authDirectives.graphql`

```
"""
```

The user must be authenticated to use the field.

```
"""
directive @private on FIELD_DEFINITION
```

Note that our focus is on field-level authorization so the only valid location for this directive is `FIELD_DEFINITION`. Next, we'll create a complimentary `authDirectives.js` file in the same directory with the following starter code:

`shared/src/directives/authDirectives.js`

```
import { dirname, resolve } from "path";
import { fileURLToPath } from "url";
import { readFileSync } from "fs";

import { ApolloError } from "apollo-server";
import { defaultFieldResolver } from "graphql";
import { getDirectives, MapperKind, mapSchema } from
  "@graphql-tools/utils";

function authDirectives() {
  // Return an object containing the auth-related directive definitions
  // plus a function that will transform Object type fields in the
  // subgraph schema to handle the directives, where applied
}

export default authDirectives;
```

In the code above, we imported the `defaultFieldResolver` from `graphql` because the code that we are about to write will wrap the resolver function for any `Object` type field where one of our authorization directives has been applied. We want to do some authorization checks before the field resolver executes, so when and if these checks pass we must manually call the resolver function inside the wrapping function. If a resolver function hasn't been provided in the executable schema, then we'll use the `default resolver`.

Additionally, we need `getDirectives` to extract any authorization directives from each field included in an operation, and we will use the `mapSchema` function and `MapperKind` to specify what element of the schema we want to transform, which are fields on `Object` types in this case.

The `authDirectives` function will return an object with an `authDirectivesTypeDefs` property that contains the SDL string of directive definitions as well as an `authDirectivesTransformer` method for a subgraph to apply to its schema so it can execute the directive handling logic:

*shared/src/directives/authDirectives.js*

```
// ...  
  
function authDirectives() {  
  const __dirname = dirname(fileURLToPath(import.meta.url));  
  
  return {  
    authDirectivesTypeDefs: readFileSync(  
      resolve(__dirname, "./authDirectives.graphql"),  
      "utf-8"  
    ),  
    authDirectivesTransformer: schema =>  
      mapSchema(schema, {  
        [MapperKind.OBJECT_FIELD]: fieldConfig => {  
          // Directive handling logic will go here...  
        }  
      })  
    },  
  };  
  
  export default authDirectives;
```

We will end up handling the logic for all of our authorization directives in this transformer function because we may need to evaluate OR relationships where multiple authorization directives are applied to a single field in some instances (for example, the `@owner` directive to authorize end users or the `@scope` directive to authorize authenticated services), but do note that defining one transformer function per custom type system directive is a common pattern.

Next, inside of the transformer function we get the directives for a field by calling `getDirectives` and then look for the `@private` directive. We then get the field's defined resolver or set the default resolver as the `resolve` value. If the `@private` directive is present, then we wrap the field resolver with some additional code that checks if there is a validated access token in the `context`. If there isn't, then we throw an error, otherwise, we proceed with executing the field resolver:

*shared/src/directives/authDirectives.js*

```
// ...  
  
function authDirectives() {  
  const __dirname = dirname(fileURLToPath(import.meta.url));  
  
  return {  
    authDirectivesTypeDefs: readFileSync(  
      resolve(__dirname, "./authDirectives.graphql"),  
      "utf-8"  
    ),  
    authDirectivesTransformer: schema =>  
      mapSchema(schema, {  
        [MapperKind.OBJECT_FIELD]: fieldConfig => {  
          // Directive handling logic will go here...  
        }  
      })  
    },  
  };  
  
  export default authDirectives;
```

```
return {
  // ...
  authDirectivesTransformer: schema =>
    mapSchema(schema, {
      [MapperKind.OBJECT_FIELD]: fieldConfig => {
        const fieldDirectives = getDirectives(schema, fieldConfig);
        const privateDirective = fieldDirectives.find(
          dir => dir.name === "private"
        );

        const { resolve = defaultFieldResolver } = fieldConfig;

        if (privateDirective) {
          fieldConfig.resolve = function (source, args, context, info) {
            const privateAuthorized = privateDirective &&
              context.user?.sub;

            if (!privateAuthorized) {
              throw new ApolloError("Not authorized!");
            }

            return resolve(source, args, context, info);
          };
        }

        return fieldConfig;
      }
    })
  );
}

export default authDirectives;
```

We'll test this code out in a moment, but first, we'll add the `@owner` directive definition to `authDirectives.graphql`. This custom directive will take a non-nullable `String` argument that indicates where in the field arguments a reference to the ID of the resource's owner may be found:

*shared/src/directives/authDirectives.graphql*

```
# ...
```

```

```
Require an authenticated user to be the owner of the resource to use the
field.

"""
directive @owner(
  """
  Name of the argument to check against the access token's `sub` claim.

  Dot notation is allowed for nested fields.
  """
  argumentName: String!
) on FIELD_DEFINITION
```

In some cases, the owner ID argument may be nested inside of the `args` parameter in the resolver, so we allow dot notation in string form to represent the relationships in the `argumentName` argument above. To facilitate searching through the resolver's `args` object using dot notation, we'll install `lodash-es`:

*shared/*

```
npm i lodash-es@4.17.21
```

Specifically, we'll need its `get` function:

*shared/src/directives/authDirectives.js*

```
// ...
import { get } from "lodash-es";
// ...

function authDirectives() {
  // ...
}

export default authDirectives;
```

Now we can build on our transformer function with logic to handle the `@owner` directive as well. The code that we add will verify that the submitted argument value matches the value of `context.user.sub`:

*shared/src/directives/authDirectives.js*

```
// ...
```

```
function authDirectives() {
  const __dirname = dirname(fileURLToPath(import.meta.url));

  return {
    // ...
    authDirectivesTransformer: schema =>
      mapSchema(schema, {
        [MapperKind.OBJECT_FIELD]: fieldConfig => {
          // ...
          const ownerDirective = fieldDirectives.find(
            dir => dir.name === "owner"
          );

          const { resolve = defaultFieldResolver } = fieldConfig;

          if (privateDirective || ownerDirective) {
            fieldConfig.resolve = function (source, args, context, info) {
              const privateAuthorized = privateDirective &&
                context.user?.sub;
              const ownerArgAuthorized =
                ownerDirective &&
                context.user?.sub &&
                get(args, ownerDirective.args.argumentName) ===
                  context.user.sub;

              if (
                (privateDirective && !privateAuthorized) ||
                (ownerDirective && !ownerArgAuthorized)
              ) {
                throw new ApolloError("Not authorized!");
              }

              return resolve(source, args, context, info);
            };
          }
        }
      });
}
```

```
export default authDirectives;
```

The `@private` and `@owner` directives are now ready to go, so we can export them from the main entry point of the shared module:

*shared/src/index.js*

```
import authDirectives from "./directives/authDirectives.js";
import DateTimeType from "./scalars/DateTimeType.js";

export { authDirectives, DateTimeType };
```

Now we're ready to import the `authDirectives` function to use in the accounts service. To do that, we must concatenate the directive definitions to the rest of the subgraph's type definitions and also apply the transformer function to the schema:

*accounts/src/index.js*

```
// ...

import { authDirectives } from "../../shared/src/index.js";
// ...

const { authDirectivesTypeDefs, authDirectivesTransformer } =
  authDirectives();
const subgraphTypeDefs = readFileSync(
  resolve(__dirname, "./graphql/schema.graphql"),
  "utf-8"
);
const typeDefs = gql(` ${subgraphTypeDefs}\n${authDirectivesTypeDefs}`);
let subgraphSchema = buildSubgraphSchema({ typeDefs, resolvers });
subgraphSchema = authDirectivesTransformer(subgraphSchema);

const server = new ApolloServer({
  schema: subgraphSchema,
  // ...
});

// ...
```

Next, we'll apply the `@private` directive to the `account` and `accounts` fields on the root `Query` type so that they may only be resolved for authenticated users:

`accounts/src/graphql/schema.graphql`

```
# ...

type Query {
  "Retrieves a single account by ID."
  account(id: ID!): Account! @private
  "Retrieves a list of accounts."
  accounts: [Account] @private
  "Retrieves the account of the currently logged-in user."
  viewer: Account
}

# ...
```

Try running one of these queries now with and without a valid JWT in the `Authorization` header. When the JWT is removed, you should see a similar error response:

*API Response*

```
{
  "errors": [
    {
      "message": "Not authorized!",
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "serviceName": "accounts",
        "exception": {
          "stacktrace": [
            "Error: Not authorized!",
            // ...
          ],
          "message": "Not authorized!",
          "locations": [
            {
              "line": 1,
              "column": 29
            }
          ],
          "path": [
            "accounts"
          ]
        }
      }
    }
}
```

```
        }
    }
],
"data": {
    "accounts": null
}
}
```

The final step for adding authorization to the accounts service will be using the `@owner` directive to ensure that an account can only be updated or deleted by the user that owns it:

`accounts/src/graphql/schema.graphql`

```
# ...

type Mutation {
    "Creates a new account."
    createAccount(input: CreateAccountInput!): Account!
    "Deletes an account."
    deleteAccount(id: ID!): Boolean! @owner(argumentName: "id")
    "Updates an account's email."
    updateAccountEmail(input: UpdateAccountEmailInput!): Account!
        @owner(argumentName: "input.id")
    "Updates an account's password."
    updateAccountPassword(input: UpdateAccountPasswordInput!): Account!
        @owner(argumentName: "input.id")
}
```

Note that we use dot notation for the nested `id` value inside of the `input` field to indicate where the account owner ID can be found in the Input Object argument for the two update mutations. Now try running these mutations again using a valid JWT obtained for a different user. Again, you should see the `Not authorized!` error returned in the response.

## Securing References Resolvers

Before we wrap up this chapter, we need to address a security hole related to accounts. As a best practice, only the gateway should be able to send requests to subgraph services in production environments. Under those conditions, bad actors shouldn't be able to query account data by writing a query that includes the `_entities` field and then send that request to the subgraph directly. However, we must consider another scenario where a request sent from the gateway may lead to unauthorized access of the accounts data.

We made the `Account` Object type an entity, which means that it can be referenced and extended in other subgraph schemas. Thinking ahead to when we add additional subgraphs, if a subgraph exposes a publicly queryable field with an Object output type that in turn references the `Account` entity as an output type for one of its fields, then there will be no authorization checks when the `__resolveReference` resolver is executed for the `Account` type in the accounts service. Here is a hypothetical example to illustrate this point:

```
type SomePublicType {
  # This field is publicly queryable
  id: ID!
  # This field should only be accessible to authenticated users
  # based on the requirements of the accounts service, but there
  # aren't any checks in place to authorize account access when
  # queried by the gateway via the `entities` field
  account: Account!
}

type Query {
  # No authorization directive applied here
  somePublicType(id: ID!): SomePublicType
}
```

Before we choose a solution to address this security issue, it's worth pausing to reflect on the relative advantages and disadvantages of the approach we just took to configure field-level authorization for the accounts service. One clear advantage to using type system directives from a shared library is that it can help standardize how authorization is handled across subgraphs, or at least for any Node.js subgraphs at the moment. This can save subgraph maintainers from having to maintain their own authorization solutions and improve predictability on the client side. However, we can't directly apply an `@private` or `@owner` directive to any of the fields that `buildSubgraphSchema` adds to our subgraph schema.

Alternatively, using resolver middleware such as GraphQL Shield or AuthZ in each subgraph may have provided more flexibility and nuance when adding field-level authorization, including reference resolvers. Additionally, resolver middleware prevents tight coupling of authorization logic to the subgraph's type definitions, which may or may not be an important schema design consideration for your project. But of course, the trade-off is a potential loss of consistency and the requirement to implement these rules on a per-subgraph basis.

We're going to stick with our directive-based approach for the Marked GraphQL API, so we have two options for authorizing access to an account via the reference resolver. The first would be to support the `OBJECT` location for the `@private` and `@owner` directives as well so that we can globally declare an authorization requirement for an Object type wherever it is used as an output type for the field. However, we will quickly run into new challenges where we need to handle fields within that object that may have more or less permissive authorization directives applied.

Additionally, applying authorization directives directly to an entire Object type such as `Account` means that we would also need some way to override it to handle a field such as `viewer` where we merely return `null` if there's no token in the context, or we would need to wrap `Account` in another type to use as an output type with this field instead.

Handling this kind of complex logic is outside the scope of this book, so we will opt for the second option which is to do one-off authorization checks directly in the reference resolvers for any entities that we define. This will be one of the key exceptions that we make when abstracting authorization logic into custom directives. Let's update the reference resolver for the `Account` type now with a few additional lines of code to check for a `user` in the context:

`accounts/src/graphql/resolvers.js`

```
import { ApolloError } from "apollo-server";

import { DateTimeType } from "../../shared/src/index.js";

const resolvers = {
  // ...
  Account: {
    __resolveReference(reference, { dataSources, user }) {
      if (user?.sub) {
        return dataSources.accountsAPI.getAccountById(reference.id);
      }
      throw new ApolloError("Not authorized!");
    },
    // ...
  },
  // ...
};

export default resolvers;
```

Apart from the `Account` type, we only need to define one other entity in the Marked GraphQL API, so this approach will be manageable for now but could be revisited and optimized as warranted in the future as additional subgraphs are added to the federated graph.

## Summary

This chapter was an opportunity to pause and make our code a bit cleaner and more developer-friendly before adding additional subgraphs to the Marked GraphQL API. We started by adding an `AccountsDataSource` to extract explicit data-fetching logic from the accounts service's resolvers.

As we move forward and write the code for the other three subgraph services, we'll add Apollo data sources from the start. We'll also reuse the `DateTime` custom Scalar type and the `@private` and `@owner` custom type system directives across the different subgraph schemas. In the next chapter, we'll build the first of the three remaining services—the profiles service.

## Chapter 4

# User Metadata Management with MongoDB and Mongoose

In this chapter, we will:

- Create a new subgraph service to contain type definitions for user profiles
- Set up MongoDB with Mongoose to store user metadata in a profiles collection
- Design the subgraph schema for a profiles service and compose it into the supergraph
- Create queries and mutations for reading and writing profile data to MongoDB
- Search users by their usernames or full names using MongoDB's full-text search
- Add authorization to various fields in the profiles service

## Why Create a Separate Service for User Metadata?

We need a way to store additional metadata about users that goes beyond their emails and passwords. Specifically, we need a way to store a user's full name, a unique user handle, interests, and a list of other users in their network.

Your first thought might be, “we can just use Auth0 for this.” If you’ve explored the Auth0 user management interface or documentation, then you may have noticed that there is a `user_metadata` field that could be used to store this extra user data. However, there are a few reasons we wouldn’t want to do this.

First, while we could technically put whatever we want in the `user_metadata` field (as well as the complementary `app_metadata` field that can be used to store read-only data about the user), Auth0 intended these fields to be used for storing information related to the users’ identities. While a user’s full name and handle seem relevant to their identity, additional details such as the bookmark topics that they are interested in and the IDs of other users in their network seem beyond this scope.

Additionally, both the `app_metadata` and `user_metadata` fields are limited to a size of 16 MB each right now, and the Auth0 documentation warns that it may put stricter limitations on the size of these fields in the future. So treating these fields like a mini database could be risky.

Finally, we need to think about the rate limits of the Auth0 Management API. If we are constantly making requests to the API to query user metadata (rather than only using it for basic CRUD operations on essential user account data), then we may find ourselves bumping into rate limits sooner than we expect.

In summary, while the metadata fields that Auth0 exposes are very handy, you don't want to overuse them. If you need to store a lot of additional metadata about users that may only be tangentially related to their identity, then it's a better idea to use a separate database because the `*_metadata` fields in Auth0 aren't designed for this purpose.

This is where MongoDB comes in. We could go a step further and store all of our user account information in MongoDB (including emails and passwords) and sync this up with our Auth0 account, but this feature is only available on paid plans. For our purposes, we can stick with our current set-up and simply use a MongoDB collection to store the additional metadata about our users. We'll include a field in a user's profile document in MongoDB that stores a reference to their Auth0 user ID to link the two related blocks of data together.

## Install Packages and Configure Mongoose

To begin, we need to create a `profiles` subdirectory in the root project directory to contain all of the code related to user profile metadata management. The `profiles` service will need its own `package.json` file:

```
profiles/
```

```
npm init --yes
```

Next, we have to install some essential packages to stand up the subgraph's GraphQL server:

```
profiles/
```

```
npm i @apollo/subgraph@2.0.3 apollo-datasource@3.3.1 apollo-server@3.7.0
dotenv@16.0.0 graphql@16.5.0
```

As well as `nodemon` to watch for file changes in the development environment:

```
profiles/
```

```
npm i -D nodemon@2.0.15
```

And again, we will update the `package.json` file to declare ES module usage in this service and add a script to start the service:

`profiles/package.json`

```
{  
  // ...  
  "type": "module",  
  "scripts": {  
    "dev": "nodemon -r dotenv/config -e env,graphql,js ./src/index.js"  
  },  
  // ...  
}
```

The type definitions in the profiles services will make use of the `DateTime` Scalar type and the authorization directives that we defined in the last chapter, so we'll run `npm link` again in the `profiles` directory to ensure that we reference the `graphql` package that was installed in shared here as well:

`profiles/`

```
npm link graphql
```

Finally, we'll add a `.env` file to `profiles` and set the `NODE_ENV` and also specify the port where this service's GraphQL endpoint will be exposed:

`profiles/.env`

```
NODE_ENV=development  
PORT=4002
```

Now can shift our attention to configuring this service's connection to MongoDB. While we could work directly with the MongoDB driver for Node.js, we're going to use [Mongoose](#) as an Object Document Mapper (ODM) instead because it provides many convenience methods for working with documents and collections. Beyond that, it also allows us to specify a schema for a collection with types for document fields. Having a typed schema will offer a measure of certainty about what's going into and coming out of our NoSQL database.

Before proceeding, please ensure you have installed MongoDB locally. Refer to the “Required Software” section in the Preface for information about installing MongoDB.

You will also need to start MongoDB before you can read and write data to a new database. For example, if you installed MongoDB on a Mac using Homebrew you can run `brew services start mongodb-community` to start up MongoDB as a macOS service.

You can find more information on how to run MongoDB in its installation documentation.

Next, we'll install the Mongoose package in `server`:

```
npm i mongoose@6.3.0
```

We'll need to set an environment variable for the URL of the database next. By default, MongoDB will be available on port 27017 (be sure to change this port number if you are running MongoDB on a different port). The `/marked-profiles` portion of the MongoDB URL is the name of the database that will be lazily created by MongoDB when we write the first document to it:

`profiles/.env`

```
MONGODB_URL=mongodb://127.0.0.1:27017/marked-profiles  
# ...
```

Now we'll create a `src` directory in `profiles` and add a `config` subdirectory to `src`. Inside of `config` we will then add a `mongoose.js` file with the following code:

`profiles/src/config/mongoose.js`

```
import mongoose from "mongoose";  
  
function initMongoose() {  
  const connectionUrl = process.env.MONGODB_URL;  
  mongoose.connect(connectionUrl);  
  
  mongoose.connection.on("connected", () => {  
    console.log(`Mongoose default connection ready at ${connectionUrl}`);  
  });  
  
  mongoose.connection.on("error", error => {  
    console.log("Mongoose default connection error:", error);  
  });  
}  
  
export default initMongoose;
```

In this file, we import `mongoose` and set up a single exported function. We'll use this function to initiate a connection with MongoDB in the profiles service. When creating the connection to MongoDB via Mongoose we have to supply the database URL as an argument, which we've referenced from the `MONGODB_URL` environment variable.

The basic Mongoose configuration is in place so we're ready to build the schema for the collection belonging to this service—the `Profile` collection. Add a subdirectory in `profiles/src` called `models` and add a `Profile.js` file to it.

To create the schema we must instantiate a new Mongoose Schema, describe all of the allowed fields for documents in this collection, and then pass the schema object itself into `mongoose.model`. The schema for `Profile` collection documents will contain `accountId`, `createdAt`, `fullName`, `interests`, `network`, and `username` fields:

`profiles/src/models/Profile.js`

```
import mongoose from "mongoose";

const profileSchema = new mongoose.Schema({
  accountId: {
    type: String,
    required: true,
    unique: true
  },
  createdAt: {
    type: Date,
    default: Date.now,
    required: true
  },
  fullName: {
    type: String,
    trim: true
  },
  interests: [String],
  network: [String],
  username: {
    type: String,
    required: true,
    trim: true,
    unique: true
  }
});

const Profile = mongoose.model("Profile", profileSchema);
```

```
export default Profile;
```

There are a few important things to note about the `Profile` document model:

- The `accountId` field will contain a reference to the associated user account in Auth0 so that we can reference and extend the `Account` entity in the profiles service's schema. Each Auth0 user account can only be associated with one related profile document, so we enforce uniqueness for this document field value.
- The `interests` field will contain a list of strings that indicate what topics the user is interested in so Marked can make relevant bookmark recommendations.
- The `network` field will contain a list of Auth0 user account IDs, which can be used to query the profiles of other users in a user's network.
- The `username` for a profile must also be unique across all documents in the collection.
- The `fullName`, `interests`, and `network` fields will be optional.

## Scaffold the Profiles Service

Now that we have declared a schema for the `Profile` collection in MongoDB we can start building out the subgraph schema for the profiles service. We'll start by creating an `index.js` file in `profiles/src` to serve as the main entry point for this service. Next, we'll create a `graphql` subdirectory in `profiles/src` with `schema.graphql` and `resolvers.js` files, plus a `dataSources` subdirectory in there with a `ProfilesDataSource.js` file.

The current file structure in `profiles` will look like this:

```
profiles
  └── node_modules/
      └── ...
  └── src/
      └── config
          └── mongoose.js
      └── graphql
          └── dataSources
              └── ProfilesDataSources.js
          └── resolvers.js
          └── schema.graphql
      └── models
          └── Profile.js
      └── index.js
  └── .env
```

```
└── package.json
└── package-lock.json
```

Our goal for this section will be to loosely wire up the new subgraph schema and compose it into the supergraph schema. Throughout the rest of this chapter, we will build out each component of our second subgraph service, including type definitions, resolvers, and a data source. First, let's set up the GraphQL server for this service in the new `index.js` file:

`profiles/src/index.js`

```
import { dirname, resolve } from "path";
import { fileURLToPath } from "url";
import { readFileSync } from "fs";

import { ApolloServer, gql } from "apollo-server";
import { buildSubgraphSchema } from "@apollo/subgraph";

import { authDirectives } from "../../shared/src/index.js";
import initMongoose from "./config/mongoose.js";
import Profile from "./models/Profile.js";
import ProfilesDataSource from
  "./graphql/dataSources/ProfilesDataSource.js";
import resolvers from "./graphql/resolvers.js";

const __dirname = dirname(fileURLToPath(import.meta.url));
const port = process.env.PORT;

const { authDirectivesTypeDefs, authDirectivesTransformer } =
  authDirectives();
const subgraphTypeDefs = readFileSync(
  resolve(__dirname, "./graphql/schema.graphql"),
  "utf-8"
);
const typeDefs = gql(`${
    subgraphTypeDefs
}\n${authDirectivesTypeDefs}`);
let subgraphSchema = buildSubgraphSchema({ typeDefs, resolvers });
subgraphSchema = authDirectivesTransformer(subgraphSchema);

const server = new ApolloServer({
  schema: subgraphSchema,
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user };
  }
});
```

```
  },
  dataSources: () => {
    return {
      profilesAPI: new ProfilesDataSource({ Profile })
    };
  }
);

initMongoose();

server.listen({ port }).then(({ url }) => {
  console.log(`Profiles service ready at ${url}`);
});
```

The code in this file is nearly identical to the code we have in the `index.js` file for the accounts service, with the exception that we import the `Profile` model (to pass into the `ProfilesDataSource` constructor) and the `initMongoose` function to connect to MongoDB from this service. Just as we did with our accounts service, we need to retrieve the user's access token to authorize API requests, so we'll set it in the Apollo Server's `context` based on the `user` header sent from the gateway.

In the `schema.graphql` file, we will only declare our intention to use the Federation 2 specification for this subgraph for now:

`profiles/src/graphql/schema.graphql`

```
extend schema
@link(url: "https://specs.apollo.dev/federation/v2.0", import: ["@key"])
```

Next, we'll add this code to the `resolvers.js` file:

`profiles/src/graphql/resolvers.js`

```
import { ApolloError, UserInputError } from "apollo-server";

const resolvers = {};

export default resolvers;
```

And this code to `ProfilesDataSource.js`:

*profiles/src/graphql/dataSources/ProfilesDataSource.js*

```
import { DataSource } from "apollo-datasource";
import { UserInputError } from "apollo-server";

class ProfilesDataSource extends DataSource {
  constructor({ Profile }) {
    super();
    this.Profile = Profile;
  }
}

export default ProfilesDataSource;
```

Back over in `gateway`, we'll add a new environment variable for the profiles service's endpoint:

*gateway/.env*

```
# ...
ACCOUNTS_ENDPOINT=http://localhost:4001
PROFILES_ENDPOINT=http://localhost:4002
```

And then update the `ApolloGateway` constructor so that it fetches the subgraph schema for the profiles service as well:

*gateway/src/config/apollo.js*

```
// ...

function initGateway(httpServer) {
  const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: "accounts", url: process.env.ACCOUNTS_ENDPOINT },
        { name: "profiles", url: process.env.PROFILES_ENDPOINT }
      ],
      pollIntervalInMs: 1000
    }),
    // ...
  });
}

// ...
```

```
}
```

```
export default initGateway;
```

All of the prerequisite plumbing is in place for the profiles service now, but it doesn't have anything to contribute to the supergraph schema just yet. To remedy that, we'll need to add some types and resolvers to the subgraph schema.

## Add the Profile Type and Its Query Fields

Now we can build out the schema for the profiles service. As we do, we'll finally have a chance to take advantage of one of Apollo Federation's killer features—namely, the ability to reference and extend entity types from other subgraphs.

We'll start by creating a new type called `Profile` in the `schema.graphql` file. Again, we'll use the `@key` directive to turn the `Profile` into an entity so that it may be referenced and extended by other subgraphs. We will also use the shared `DateTime` Scalar type for the `createdAt` field in the `Profile` type, so we have to redefine it in this subgraph as well:

`profiles/src/graphql/schema.graphql`

```
# ...
```

```
"""
An ISO 8601-encoded UTC date string.
"""

scalar DateTime

"""
A profile contains metadata about a specific user.
"""

type Profile @key(fields: "id") {
  "The unique ID of the user's profile."
  id: ID!
  "The date and time the profile was created."
  createdAt: DateTime!
  "The Auth0 account tied to this profile."
  account: Account!
  "The full name of the user."
  fullName: String
  "A tag-like list of topics of interest to the user."
  interests: [String]
```

```
  "The unique username of the user."
  username: String!
}
```

We want each user profile to be connected to its corresponding account, so we add an `account` field and set the type as `Account`. Similarly, we also want to be able to traverse the graph in the other direction and retrieve the profile data from within the context of an `Account` object. Thankfully, the Apollo Federation makes it easy to define this relationship as well.

To use `Account` as the output type for the `account` field, we must now define a stub of the `Account` Object type here as well:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
An account is a unique Auth0 user.
"""

type Account @key(fields: "id") {
  id: ID!
}

# ...
```

The stub definition will just include the `id` field from the original type definition in the accounts service—this is the only piece of account data that the profiles service is aware of (stored in the MongoDB document's `accountId` field) and it's also the piece of data that other subgraphs need to have if they wish to reference or extend the `Account` entity, as defined by its `@key` directive.

If we were only going to reference the entity in this profiles service's subgraph schema without contributing any fields to it, we would also set an additional argument on the `@key` directive here of `resolvable: false` to indicate that this subgraph doesn't define a reference resolver for this type. However, as noted previously, we want to be able to connect user profiles to their relevant account data and vice versa, so we will leave out that argument and extend the `Account` type with an additional `profile` field here. Note that the `profile` field will be nullable because the Auth0 user account is always created first and there may not be a profile associated with it until some time afterward:

```
# ...

"""
An account is a unique Auth0 user.
```

```
"""
type Account @key(fields: "id") {
  id: ID!
  "Metadata about the user that owns the account."
  profile: Profile
}

# ...
```

The code that we just wrote to reference and extend the `Account` entity in this subgraph schema may seem mundane, but it's really quite magical and it illustrates one of the key value propositions of Apollo Federation. We can design a distributed GraphQL schema and we don't have to settle for drawing awkward boundaries between subgraphs based on types alone. Instead, each subgraph team can define types that other subgraphs may use and build upon as long as they know the `@key` field values for the underlying resources. The gateway will take care of retrieving and assembling that data so that it appears to the client that it came from a single data source. Very powerful stuff!

What's more, we will see in a moment that we won't need to go into the accounts service's `resolvers.js` file to write the resolver for the new `profile` field in the `Account` type (because there would be no way for the accounts service to know about profile documents without access to MongoDB). Instead, we'll write the resolver for this new `Account` field in the profiles service itself, which will allow us to maintain clear boundaries between the services and the unique data sources that they access.

The final addition to the schema in this section will be standard queries for retrieving a list of profiles or a single profile by its ID:

`profiles/src/graphql/schema.graphql`

```
# ...

type Query {
  "Retrieves a single profile by username."
  profile(username: String!): Profile! @private
  "Retrieves a list of profiles."
  profiles: [Profile] @private
}
```

Note that both `Query` fields may only be requested by an authenticated user, so we have applied the `@private` directive to both. Now we can begin building out our data source in `ProfilesDataSource.js`. We'll start with a `getProfile` method:

profiles/src/graphql/dataSources/ProfilesDataSource.js

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    getProfile(filter) {
        return this.Profile.findOne(filter).exec();
    }
}

export default ProfilesDataSource;
```

We'll pass an object (also known as a *query document*) into Mongoose's `findOne` method where its value is set to the `filter` object parameter. Generalizing the `filter` will allow us to query a profile document based on various fields such as `accountId` or `username`. Lastly, we chain the `exec` method on the end so Mongoose will return a fully-fledged Promise.

Next, we will create a `getProfiles` method using Mongoose's `find` method:

profiles/src/graphql/dataSources/ProfilesDataSource.js

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    getProfiles() {
        return this.Profile.find({}).exec();
    }
}

export default ProfilesDataSource;
```

Passing `{}` into the `find` method will return all the documents in the collection, rather than filtering them based on some selection criteria. Now we can add resolvers for the `profile` and `profiles` queries in `resolvers.js`, as well as the `DateTime` resolver:

profiles/src/graphql/resolvers.js

```
import { ApolloError, UserInputError } from "apollo-server";
```

```
import { DateTimeType } from "../../shared/src/index.js";

const resolvers = {
  DateTime: DateTimeType,

  Query: {
    async profile(root, { username }, { dataSources }) {
      const profile = await dataSources.profilesAPI.getProfile({ username });

      if (!profile) {
        throw new UserInputError("Profile not available.");
      }
      return profile;
    },
    profiles(root, args, { dataSources }) {
      return dataSources.profilesAPI.getProfiles();
    }
  }
};

export default resolvers;
```

The data source's `getProfile` method returns the value of Mongoose's `findOne` method, and `findOne` will return `null` if no matching document is found. For this reason, we throw a `UserInputError` with an instructive error message in the `profile` resolver if no matching profile was found because the `profile` query expects a non-nullable `Profile` type to be returned.

Before moving on, we have to address the issue of resolving the `profile` field for the extended `Account` type, as well as resolving the `id` and `account` fields for the `Profile` type. To resolve the `profile` field on `Account`, we need to use an Auth0 account ID to look up the correct user profile. The only piece of data that connects Auth0 user accounts and MongoDB profile documents is the Auth0 account ID in the `accountId` document field, so we have to look up the profile document based on this field value. We can reuse the `getProfile` method from the `ProfilesDataSource` again to do this:

`profiles/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  DateTime: DateTimeType,
```

```
Account: {
  profile(account, args, { dataSources }) {
    return dataSources.profilesAPI.getProfile({
      accountId: account.id
    });
  }
},
// ...
};

export default resolvers;
```

Next, we need to resolve the `id`, and `account` fields on `Profile` because there aren't any directly corresponding field names in the profile documents so we can't rely on the default resolver here. We also need to write the reference resolver for `Profile` so that the profiles service knows how to resolve this entity when referenced by another subgraph.

We said that the key for the profile entity would be the `_id` of its MongoDB document because we know that this value is unique and will not change. While the `username` field is unique, users will be allowed to update their usernames after they create an account if desired, so the `username` will not be a stable value to reference.

We'll write a dedicated method to get a profile document by its ID in the `ProfilesDataSource` class using the Mongoose's `findById` method:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
  // ...

  getProfileById(id) {
    return this.Profile.findById(id).exec();
  }

  // ...
}

export default ProfilesDataSource;
```

Using these new data source methods we can create all of the `Profile` field resolvers now:

profiles/src/graphql/resolvers.js

```
// ...

const resolvers = {
  // ...

  Profile: {
    __resolveReference(reference, { dataSources, user }) {
      if (user?.sub) {
        return dataSources.profilesAPI.getProfileById(reference.id);
      }
      throw new ApolloError("Not authorized!");
    },
    account(profile) {
      return { id: profile.accountId };
    },
    id(profile) {
      return profile._id;
    }
  },
  // ...
};

export default resolvers;
```

Take note that resolving the `account` field on `Profile` is different from a typical resolver. Again, the profiles service doesn't know much about user accounts because its main concern is the profile data stored in MongoDB. For that reason, we can't write a field resolver for the `account` field as we normally would. The solution is to return a representation of the external `Account` entity based on its `id` key. That means we will set the `id` property in the returned object to the `profile.accountId` value because `id` is the key field we previously determined that we were going to use to connect the `Account` entity to types in other subgraphs. We should now be able to start up the profiles service from a new terminal tab without errors:

profiles/

```
npm run dev
```

While the service is running now, we still have an outstanding issue to resolve—we don't have any profile documents in MongoDB so we can't test out the new Query fields just yet.

## Add a `createProfile` Mutation

To test the resolvers we just wrote we need a `createProfile` mutation to add a new document to MongoDB that contains the user metadata for an associated Auth0 account. We'll start by defining an Input Object type that will be used as an argument for the mutation:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
Provides data to create a new user profile.
"""


```

To create a new user profile, we must provide the Auth0 account ID and a unique username at a minimum. The user can optionally set some interests and their full name. We won't concern ourselves with setting anything initially in the `network` field that was defined in the Mongoose `Profile` model because new users will only be able to follow other users after the account and profile creation process is complete.

Now we can add a `createProfile` field on the root `Mutation` type. Users can only create new profiles for their accounts, so we will use the `@owner` directive to confirm that the `accountId` submitted with the mutation matches the user's `sub` claim in an authenticated user's token:

`profiles/src/graphql/schema.graphql`

```
# ...

type Mutation {
  "Creates a new profile tied to an Auth0 account."
  createProfile(input: CreateProfileInput!): Profile!
```

```
    @owner(argumentName: "input.accountId")  
}
```

Next, we'll shift our attention back to the `ProfilesDataSource` class. Whenever we save a user's list of interests to the database, we want to make sure that each string in the list is in a tag-like format where the letter characters are all lowercase and any spaces used for whitespace are replaced with hyphens. We will add a `_formatTags` method to support that requirement and then use it to format the `interests` field during user profile creation in a new `createProfile` method:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...  
  
class ProfilesDataSource extends DataSource {  
    // ...  
  
    _formatTags(tags) {  
        return tags.map(tag => tag.replace(/\s+/g, "-").toLowerCase());  
    }  
  
    createProfile(profile) {  
        if (profile.interests) {  
            const formattedTags = this._formatTags(profile.interests);  
            profile.interests = formattedTags;  
        }  
  
        const newProfile = new this.Profile(profile);  
        return newProfile.save();  
    }  
  
    // ...  
}  
  
export default ProfilesDataSource;
```

Now we can add a `createProfile` resolver:

`profiles/src/graphql/resolvers.js`

```
// ...  
  
const resolvers = {
```

```
// ...
Query: {
  // ...
},
Mutation: {
  createProfile(root, { input }, { dataSources }) {
    return dataSources.profilesAPI.createProfile(input);
  }
}
};

export default resolvers;
```

The `createProfile` mutation is now ready for testing Explorer. You'll need to get the account ID of an existing user in Auth0 to use in the `input` argument. Make sure that you submit a valid access token in the `Authorization` header too when running this mutation:

#### *GraphQL Mutation*

```
mutation CreateProfile($input: CreateProfileInput!) {
  createProfile(input: $input) {
    id
    createdAt
    username
    interests
    fullName
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "accountId": "auth0|625b5a7847a7f7006f3ce7ab",
    "fullName": "Sir Marks-a-Lot",
    "interests": ["graphql"],
    "username": "marksalot"
  }
}
```

*API Response*

```
{  
  "data": {  
    "createProfile": {  
      "id": "625b5bbba1dad7f5f2eaa30d",  
      "createdAt": "2022-04-17T00:13:47.845Z",  
      "username": "marksalot",  
      "interests": [  
        "graphql"  
      ],  
      "fullName": "Sir Marks-a-Lot"  
    }  
  }  
}
```

If you like, you can verify that the document ended up where you expected in MongoDB by running the `mongo` command to enter the MongoDB shell and then query the document as follows:

```
mongo  
> use marked-profiles  
> db.profiles.find()
```

You should see the following output:

```
{ "_id" : ObjectId("625b5bbba1dad7f5f2eaa30d"), "accountId" :  
  "auth0|625b5a7847a7f7006f3ce7ab", "fullName" : "Sir Marks-a-Lot",  
  "interests" : [ "graphql" ], "network" : [ ], "username" : "marksalot",  
  "createdAt" : ISODate("2022-04-17T00:13:47.845Z"), "__v" : 0 }
```

You can also use an application such as [MongoDB Compass](#) to browse the documents that have been added to the profiles collection.

With a profile in the database, we also test out the profile-related queries we previously wrote. First, we'll test the `profiles` query:

*GraphQL Query*

```
query Profiles {  
  profiles {
```

```
    id
    account {
      id
      email
    }
    fullName
    username
  }
}
```

#### API Response

```
{
  "data": {
    "profiles": [
      {
        "id": "625b5bbba1dad7f5f2eaa30d",
        "account": {
          "id": "auth0|625b5a7847a7f7006f3ce7ab",
          "email": "marksalot@markedmail.com"
        },
        "fullName": "Sir Marks-a-Lot",
        "username": "marksalot"
      }
    ]
  }
}
```

Next we'll test the `profile` query, searching the specific user by `username`. We'll also include the `account` field with a sub-selection of `Account` fields to confirm that data can be resolved across both subgraphs in a single query:

#### GraphQL Query

```
query Profile($username: String!) {
  profile(username: $username) {
    id
    account {
      id
      email
    }
    fullName
```

```
    username  
}  
}
```

### Query Variables

```
{  
  "username": "marksalot"  
}
```

### API Response

```
{  
  "data": {  
    "profile": {  
      "id": "625b5bbba1dad7f5f2eaa30d",  
      "account": {  
        "id": "auth0|625b5a7847a7f7006f3ce7ab",  
        "email": "marksalot@markedmail.com"  
      },  
      "fullName": "Sir Marks-a-Lot",  
      "username": "marksalot"  
    }  
  }  
}
```

We can also test the `accounts` query again to see if we get profile information from it now:

### GraphQL Query

```
query Accounts {  
  accounts {  
    id  
    email  
    profile {  
      username  
    }  
  }  
}
```

*API Response*

```
{
  "data": {
    "accounts": [
      {
        "id": "auth0|625b5a7847a7f7006f3ce7ab",
        "email": "marksalot@markedmail.com",
        "profile": {
          "username": "marksalot"
        }
      }
    ]
  }
}
```

Again, it's quite powerful that, from a client's perspective, all of this data can be queried from a single GraphQL endpoint and in a way that reflects the natural relationships between the nodes in the graph. At the same time, clients never have to concern themselves that separate back-end teams may manage the `Account` and `Profile` type definitions independently.

What happens under the hood in the gateway to resolve this data isn't a black box either. An `ApolloGateway` can be configured to log out its *query plans* by setting its `debug` option to `true`, as illustrated below:

`gateway/src/config/apollo.js`

```
// ...

function initGateway(httpServer) {
  const gateway = new ApolloGateway({
    debug: true,
    // ...
  });

  // ...
}

export default initGateway;
```

After rerunning the previous `accounts` query, we can see exactly what subgraphs are involved in query plan execution, what fields are resolved by which subgraph, and how many network hops are required for an operation:

```
QueryPlan {  
  Sequence {  
    Fetch(service: "accounts") {  
      accounts {  
        __typename  
        id  
        email  
      }  
    },  
    Flatten(path: "accounts.@") {  
      Fetch(service: "profiles") {  
        ... on Account {  
          __typename  
          id  
        }  
      } =>  
      {  
        ... on Account {  
          profile {  
            username  
          }  
        }  
      },  
    },  
  },  
}
```

The syntax above may seem a bit dense at first, but if we dig in we can see the gateway's first step is to query the accounts service for the `id` and `email` data for each account. Once that data is resolved, the next step is to query the profiles service to fetch the usernames for each user based on the account IDs. As we can see, query plans can be a helpful tool for understanding how an operation is executed across subgraphs and can be very useful for debugging and performance optimization purposes.

You can learn more about [query plan structure and output](#) in the Apollo Federation documentation.

## Add `updateProfile` and `deleteProfile` Mutations

We can create profiles now, but we still don't have any way to update or delete them yet. Unlike account updates where a user would only update their email address or password per mutation, profile fields will be simultaneously updated through the Marked user interface. For that reason, we will create an `updateProfile` mutation that can handle multiple fields. First, we'll create an Input Object type to use as an argument in an `updateProfile` mutation:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
Provides data to update an existing profile.
"""


```

Note that we could uniquely identify a profile based on its `accountId`, `id`, or `username` fields, but we'll opt for the `accountId` here to facilitate the authorization check before the mutation operation runs. Now we'll add the `updateProfile` mutation itself:

`profiles/src/graphql/schema.graphql`

```
# ...

type Mutation {
  # ...
  "Updates a user's profile details."
  updateProfile(input: UpdateProfileInput!): Profile!
    @owner(argumentName: "input.accountId")
}
```

Before creating this mutation's resolver we'll add a method to the `ProfilesDataSource` class that will update the profile document in MongoDB:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
  // ...

  async updateProfile(accountId, updatedProfileData) {
    if (
      !updatedProfileData ||
      (updatedProfileData && Object.keys(updatedProfileData).length === 0)
    ) {
      throw new UserInputError("You must supply some profile data to
        update.");
    }

    if (updatedProfileData.interests) {
      const formattedTags = this._formatTags(updatedProfileData.interests);
      updatedProfileData.interests = formattedTags;
    }

    return await this.Profile.findOneAndUpdate(
      { accountId },
      updatedProfileData,
      {
        new: true
      }
    ).exec();
  }

  export default ProfilesDataSource;
```

In this method, we require that the user's `accountId` is always passed as the first argument so we can use it to find the correct user profile document to update in MongoDB. All other fields are optional, but we'll throw an error if no fields were included. Also, note that we mark the method as `async` and use the `await` keyword with `findOneAndUpdate` so we don't get an error from Mongoose about `duplicate query execution` if we run the query more than once.

Note that the first argument passed to `findOneAndUpdate` is an object containing the fields needed to find a matching document (in our case, only the unique `accountId` field). The second

argument is an object containing the fields we wish to update. By passing `{ new: true }` as the final argument, this method will return the updated document once the new field values are successfully saved.

Now we can add the `updateProfile` mutation in `resolvers.js`:

`profiles/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Mutation: {
    createProfile(parent, { input }, { dataSources }) {
      return dataSources.profilesAPI.createProfile(input);
    },
    updateProfile(root, { input: { accountId, ...rest } }, { dataSources }) {
      return dataSources.profilesAPI.updateProfile(accountId, rest);
    }
  }
};

export default resolvers;
```

Let's test out the new mutation in Explorer:

*GraphQL Query*

```
mutation UpdateProfile($input: UpdateProfileInput!) {
  updateProfile(input: $input) {
    interests
    username
  }
}
```

*Mutation Variables*

```
{
  "input": {
    "accountId": "auth0|625b5a7847a7f7006f3ce7ab",
    "interests": ["javascript", "graphql"]
```

```
    }
}
```

*API Response*

```
{
  "data": {
    "updateProfile": {
      "interests": [
        "javascript",
        "graphql"
      ],
      "username": "marksalot"
    }
  }
}
```

We'll run through the same series of steps to add the `deleteProfile` mutation now. First, we'll add the `deleteProfile` mutation to the schema in `schema.graphql`:

`profiles/src/graphql/schema.graphql`

```
# ...

type Mutation {
  # ...
  "Deletes a user profile."
  deleteProfile(accountId: ID!): Boolean!
    @owner(argumentName: "input.accountId")
  # ...
}
```

Next, we'll add a related data source method in `ProfilesDataSource.js`:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
  // ...

  async deleteProfile(accountId) {
```

```
try {
  await this.Profile.findOneAndDelete({
    accountId
  }).exec();
  return true;
} catch {
  return false;
}
}

export default ProfilesDataSource;
```

Again, for deletion, we'll make the method `async` so we can wait for the promise to resolve before returning a boolean to indicate whether the operation was successful. Lastly, we'll add the `deleteProfile` resolver to `resolvers.js`:

*profiles/src/graphql/resolvers.js*

```
// ...

const resolvers = {
  // ...

  Mutation: {
    // ...
    deleteProfile(root, { accountId }, { dataSources }) {
      return dataSources.profilesAPI.deleteProfile(accountId);
    },
    // ...
  }
};

export default resolvers;
```

Now we can test out the new mutation in Explorer:

*GraphQL Mutation*

```
mutation DeleteProfile($accountId: ID!) {
  deleteProfile(accountId: $accountId)
}
```

*Mutation Variables*

```
{  
  "accountId": "auth0|625b5a7847a7f7006f3ce7ab"  
}
```

*API Response*

```
{  
  "data": {  
    "deleteProfile": true  
  }  
}
```

## Add and Remove Users from Another User's Network

Using Marked will be more interesting for users if there's a way to discover other people and their bookmarks, so the last two mutations for this schema will allow users to find and establish a connection with other users. To complete the next section, you'll need at least two user accounts set up in Auth0 with two associated user profiles in MongoDB. Use the `createAccount` and `createProfile` mutations in Explorer to create the accounts and profiles if you haven't done so already.

First, we'll add network-related fields to the `Profile` type called `network` and `isInNetwork`:

`profiles/src/graphql/schema.graphql`

```
# ...  
  
"""  
A profile contains metadata about a specific user.  
"""  
type Profile @key(fields: "id") {  
  # ...  
  "Whether the currently authenticated user has another user in their  
  network."  
  isInNetwork: Boolean  
  "Other users that have been added to the user's network."  
  network: [Profile]  
  # ...  
}
```

```
# ...
```

The `network` field provides a list of the member profiles of a user's network and the `isInNetwork` field will indicate whether an authenticated user is viewing another user that they have already added to their network (which can be helpful when rendering user interface elements in a client application). A more advanced version of the Marked application could also allow users to collaborate on collections of bookmarks with other users in their network.

Now we'll address the network-related mutations. Both of these mutations are very similar, so we can tackle both of them at the same time. We'll start by updating the schema with a new Input Object:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
Provides the unique ID of an existing profile to add or remove from a
network.
"""


```

The `networkMemberId` field will do two jobs for us. In the context of adding a user to a network, it will refer to the user to be added. Alternatively, when running a mutation to remove a user from another user's network, it will refer to the user being removed. We will keep track of network members by their Auth0 IDs because these are stable values and this approach will make it easier to clean up all references to a deleted user later on in Chapter 9.

Now we'll add `addToNetwork` and `removeFromNetwork` mutations to the schema using the `NetworkMemberInput` as an argument for both. Users will only be able to update their own networks, so we'll add the `@owner` directive as well:

`profiles/src/graphql/schema.graphql`

```
# ...

type Mutation {
  "Allows one user to add another user to their network."
  addToNetwork(input: NetworkMemberInput!): Profile!
    @owner(argumentName: "input.accountId")
  # ...
  "Allows one user to remove another user from their network."
  removeFromNetwork(input: NetworkMemberInput!): Profile!
    @owner(argumentName: "input.accountId")
  # ...
}

# ...
```

Next, we'll have to update the `ProfilesDataSource` class with a few new methods to support these mutations. First, we'll add two methods that update a profile's `network` field with the ID of the user they wish to add or remove as a member of their network:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
  // ...

  async addToNetwork(accountId, accountIdToFollow) {
    if (accountId === accountIdToFollow) {
      throw new UserInputError("User cannot be added to their own
        network.");
    }

    return await this.Profile.findOneAndUpdate(
      { accountId },
      { $addToSet: { network: accountIdToFollow } },
      { new: true }
    ).exec();
  }

  async removeFromNetwork(accountId, accountIdToFollow) {
    return await this.Profile.findOneAndUpdate(
```

```
        { accountId },
        { $pull: { network: accountIdToFollow } },
        { new: true }
    ).exec();
}

// ...

}

export default ProfilesDataSource;
```

Note the use of the `$addToSet` operator in the `addToNetwork` method. By using this operator instead of `$push`, we can be certain that all values added to this array field will be unique (because it wouldn't make sense to follow the same user twice). Similarly, we use the `$pull` operator in the `removeFromNetwork` method to delete the matching ID from the `network` array field.

Now we'll add a method to fetch all of the profile documents for members of a user's network:

*profiles/src/graphql/dataSources/ProfilesDataSource.js*

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    getNetworkProfiles(network) {
        return this.Profile.find({ accountId: { $in: network } }).exec();
    }

    // ...
}

export default ProfilesDataSource;
```

Lastly, we need another method to support the `isInNetwork` field that will check if the authenticated user has another user in their network:

*profiles/src/graphql/dataSources/ProfilesDataSource.js*

```
// ...

class ProfilesDataSource extends DataSource {
    // ...
```

```
async checkViewerHasInNetwork(viewerAccountId, accountId) {
  const viewerProfile = await this.Profile.findOne({
    accountId: viewerAccountId
  })
    .select("network")
    .exec();

  return viewerProfile.network.includes(accountId);
}

// ...
}

export default ProfilesDataSource;
```

Over in `resolvers.js`, we'll first add resolvers for the `network` and `isInNetwork` fields for the `Profile` type:

`profiles/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Profile: {
    // ...
    network(profile, args, { dataSources }) {
      return dataSources.profilesAPI.getNetworkProfiles(profile.network);
    },
    id(profile) {
      return profile._id;
    },
    isInNetwork(profile, args, { dataSources, user }) {
      return dataSources.profilesAPI.checkViewerHasInNetwork(
        user.sub,
        profile.accountId
      );
    }
  },
};

export default resolvers;
```

```
// ...
};

export default resolvers;
```

We'll also create the resolvers for the `addToNetwork` and `removeFromNetwork` mutations:

*profiles/src/graphql/resolvers.js*

```
// ...

const resolvers = {
  // ...

  Mutation: {
    addToNetwork(
      root,
      { input: { accountId, networkMemberId } },
      { dataSources }
    ) {
      return dataSources.profilesAPI.addToNetwork(accountId,
        networkMemberId);
    },
    // ...
    removeFromNetwork(
      root,
      { input: { accountId, networkMemberId } },
      { dataSources }
    ) {
      return dataSources.profilesAPI.removeFromNetwork(
        accountId,
        networkMemberId
      );
    },
    // ...
  }
};

export default resolvers;
```

Now we can try out the mutations in Explorer. First, we'll add a user to another user's network:

*GraphQL Mutation*

```
mutation AddToNetwork($input: NetworkMemberInput!) {
  addToNetwork(input: $input) {
    fullName
    username
    account {
      id
    }
    network {
      fullName
      username
      account {
        id
      }
    }
  }
}
```

*Mutation Variables*

```
{
  "input": {
    "accountId": "auth0|625b5a7847a7f7006f3ce7ab",
    "networkMemberId": "auth0|625c652808eac7006851b39f"
  }
}
```

*API Response*

```
{
  "data": {
    "addToNetwork": {
      "fullName": "Sir Marks-a-lot",
      "username": "marksalot",
      "account": {
        "id": "auth0|625b5a7847a7f7006f3ce7ab"
      },
      "network": [
        {
          "fullName": "Marky Marked",
          "username": "funkybunch",
        }
      ]
    }
  }
}
```

```
        "account": {
          "id": "auth0|625c652808eac7006851b39f"
        }
      ]
    }
}
```

Now if we re-run the `profiles` query while still authenticated as the user that ran the mutation, we can see that the `isInNetwork` value for their network member is `true`:

*GraphQL Query*

```
query Profiles {
  profiles {
    username
    isInNetwork
  }
}
```

*API Response*

```
{
  "data": {
    "profiles": [
      {
        "username": "marksalot",
        "isInNetwork": false
      },
      {
        "username": "funkybunch",
        "isInNetwork": true
      }
    ]
  }
}
```

We can try removing that user from their network as well:

*GraphQL Mutation*

```
mutation RemoveFromNetwork($input: NetworkMemberInput!) {
  removeFromNetwork(input: $input) {
    fullName
    username
    account {
      id
    }
    network {
      fullName
      username
      account {
        id
      }
    }
  }
}
```

*Mutation Variables*

```
{
  "input": {
    "accountId": "auth0|625b5a7847a7f7006f3ce7ab",
    "networkMemberId": "auth0|625c652808eac7006851b39f"
  }
}
```

*API Response*

```
{
  "data": {
    "removeFromNetwork": {
      "fullName": "Sir Marks-a-lot",
      "username": "marksalot",
      "account": {
        "id": "auth0|625b5a7847a7f7006f3ce7ab"
      },
      "network": []
    }
  }
}
```

## Add a Search Query with Full-Text Search in MongoDB

We can add and remove other Marked users as network members now, but it would be helpful if there was a way to search for these other users in the first place! As a finishing touch to the profiles service, we’re going to add a special query that allows users to search for other users by their full name or username. We’ll need to take advantage of a feature of MongoDB that allows *full-text search* by using a text index to implement this query.

Full-text search works by searching a *full-text database* for a specific query. A full-text database contains textual documents, rather than metadata alone. While this might seem like overkill when simply searching for usernames and full names, there are some useful features built into MongoDB’s full-text search that we’ll want to take advantage of here, such as filtering stop words, stemming words, and results scoring.

Full-text search in MongoDB is a great option for our app because it’s more performant, flexible, and forgiving than other naive approaches to searching through text strings (such as using regex). Using the built-in full-text search will be more than sufficient to satisfy our requirements for Marked, and it will also save us the time and effort of implementing a heavier external solution like Elasticsearch. What’s more, implementing full-text search with MongoDB is surprisingly straightforward.

But alas, there’s no such thing as a performance free lunch! Full-text search in MongoDB requires us to create a text index, which will have a performance penalty when new documents are inserted. However, this particular performance hit should be relatively small and worth the trade-off for the convenience and effectiveness of using MongoDB’s built-in search mechanism.

Before we can update the GraphQL schema to add a search query, we need to update the Mongoose `Profile` model to create a text index. In MongoDB, each collection can only have one text index, but that index can contain multiple fields. We’ll add the `fullName` and `username` fields to the text index for the `profiles` collection.

In the `Profile.js` file, add the following line of code after instantiating the `Profile` schema:

`profiles/src/models/Profile.js`

```
// ...

profileSchema.index({ fullName: "text", username: "text" });

const Profile = mongoose.model("Profile", profileSchema);

export default Profile;
```

If we restart the profiles service right now, we can see that our index has been created when we browse to the “Indexes” tab in MongoDB Compass:

MongoDB Compass - localhost:27017/marked-profiles.profiles

My Cluster

localhost:27017 STANDALONE

**marked-profiles.profiles**

Documents Schema Explain Plan **Indexes** Validation

**CREATE INDEX**

| Name and Definition         | Type    | Size    | Usage                   | Properties | Drop |
|-----------------------------|---------|---------|-------------------------|------------|------|
| _id_                        | REGULAR | 36.0 KB | 0 since Sat Apr 16 2022 | UNIQUE     |      |
| fullName_text_username_text | TEXT    | 20.0 KB | 0 since Sun Apr 17 2022 | COMPOUND   |      |
| username_1                  | REGULAR | 36.0 KB | 5 since Sat Apr 16 2022 | UNIQUE     |      |

If we were to change anything about this text index in the future (for example, add another field to it), then we would need to delete the existing text index here so it can be recreated on the server restart. In the profiles service's type definitions, we will add a new field on the root `Query` type to support user profile search:

`profiles/src/graphql/schema.graphql`

```
# ...

type Query {
  # ...
  "Performs a search of user profiles. Results are available in descending
   order by relevance only."
  searchProfiles(
    "The text string to search for in usernames or full names."
    query: String!
  ): [Profile]
}

# ...
```

In `ProfilesDataSource.js`, we'll add a new `searchProfiles` method:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    searchProfiles(searchString) {
        return this.Profile.find(
            { $text: { $search: searchString } },
            { score: { $meta: "textScore" } }
        )
            .sort({ score: { $meta: "textScore" }, _id: -1 })
            .exec();
    }

    // ...
}

export default ProfilesDataSource;
```

There are a few things to note in the code above. The first is the use of the `$text` query operator. The string of text we search for may contain multiple words, so this operator tokenizes the string value of the `$search` field using whitespace and punctuation delimiters and then performs a logical OR on all of the tokens. In other words, MongoDB will look through the applicable fields in the profile documents to find instances of any of the words in the search string.

The second thing to note is that we pass in a *projection* as a second argument so that the `textScore` is included in the returned document. We want to include the `textScore` so that we can sort on it later—meaning that we will sort the matched documents based on the relevance score MongoDB assigns them. For our search-sorting purposes, the sort expression that we use to sort the search results (using the chained `sort` method) looks like this:

```
{ score: { $meta: "textScore" }, _id: -1 }
```

Again, `{ $meta: "textScore" }` sorts the documents in descending order by relevance, and we add a criterion of `id: -1` to further sort the scored documents by their IDs afterward. The secondary `_id`-based sort will be necessary to successfully implement cursor-based pagination in the next chapter.

Lastly, we'll call the new data source method inside of the `searchProfiles` resolver:

*profiles/src/graphql/resolvers.js*

```
// ...  
  
const resolvers = {  
  // ...  
  
  Query: {  
    // ...  
    profiles(root, args, { dataSources }) {  
      return dataSources.profilesAPI.getProfiles();  
    },  
    searchProfiles(root, { query }, { dataSources }) {  
      return dataSources.profilesAPI.searchProfiles(query);  
    }  
  },  
  // ...  
};  
  
export default resolvers;
```

With the resolver in place, we can now test the new operation. Try searching for a user by username:

*GraphQL Query*

```
query SearchProfiles($query: String!) {  
  searchProfiles(query: $query) {  
    fullName  
    username  
  }  
}
```

*Query Variables*

```
{  
  "query": "marksalot"  
}
```

*API Response*

```
{  
  "data": {  
    "searchProfiles": [  
      {  
        "fullName": "Sir Marks-a-Lot",  
        "username": "marksalot"  
      }  
    ]  
  }  
}
```

Try updating the query variables for the same search operation to look for two users at once, using a mix of username and full name strings:

*GraphQL Query*

```
{  
  "query": "Sir funkybunch"  
}
```

*API Response*

```
{  
  "data": {  
    "searchProfiles": [  
      {  
        "fullName": "Marky Marked",  
        "username": "funkybunch"  
      },  
      {  
        "fullName": "Sir Marks-a-Lot",  
        "username": "marksalot"  
      }  
    ]  
  }  
}
```

## Summary

We have now built out the majority of the profiles service. We set up MongoDB and used Mongoose as an ODM to create a schema for the profiles collection. Using the data in the database's profiles collection, we built out a second subgraph and composed it into the supergraph. We can now support a variety of queries and mutations in this service, including a search query to retrieve user profiles by username or full name. We also had an opportunity to reuse the `DateTime` Scalar type and authorization directives from the shared library that we created in Chapter 3.

In the next chapter, we'll augment the profiles service with pagination—an important feature when dealing with large lists of data such as the collection of user profiles.

## Chapter 5

# Relay-Style Pagination

In this chapter, we will:

- Explore Relay-style pagination and its algorithms
- Create a `Pagination` class to implement Relay-style pagination on documents fetched from MongoDB
- Add pagination to the `network` field on the `Profile` type
- Add pagination to the `profiles` and `searchProfiles` queries

## Pagination Primer

As far as “good problems to have” go, at some point, we would hope that enough users are active on Marked that it will no longer be feasible to retrieve the entire list of user profiles for the `profiles` or `searchProfiles` queries or the full list of `network` members associated with a profile all at once. This is where pagination comes in. From now on, where long lists of data may be queried we will limit the number of results initially retrieved from MongoDB, but allow users to fetch more pages of results with subsequent requests.

While pagination can be easy to take for granted in the apps we use every day, it is often one of the trickiest things to reason about and implement properly. Relay-style pagination is no exception, and it requires additional considerations to ensure a GraphQL schema adheres to Relay’s opinionated specification.

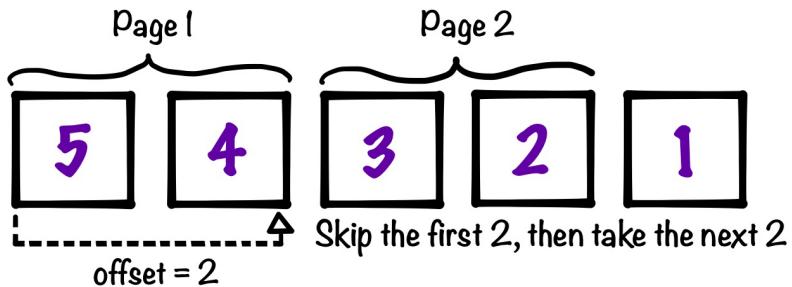
Relay-style pagination certainly isn’t the only option for paginating results with a GraphQL API either. Before we jump into implementing pagination in Marked, let’s explore different styles of pagination that we could consider using here.

### Offset-Based

Historically, offset-based pagination has been a popular choice for paginating results from a database. With offset-based pagination, a client provides information about the number of results

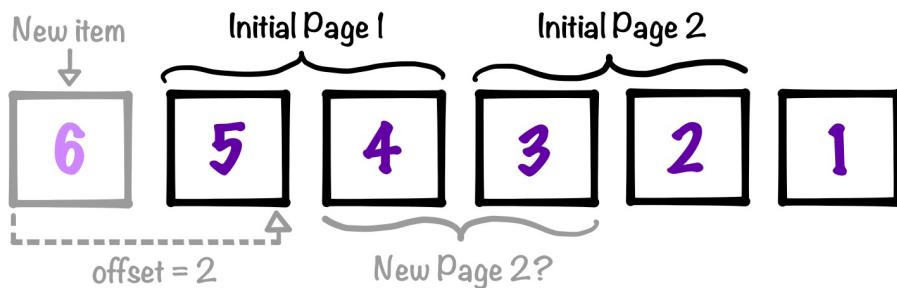
it wants to receive per page (called the *limit*) and how many results to skip before retrieving the limited number of items (called the *offset*). The server uses these criteria to query the database for that specific set of results (setting a default limit and offset, if necessary).

To visualize how offset-based pagination works, imagine you have a dataset with five items in it and you want to retrieve the second page of those items sorted in descending order with a limit of two items per page:



Offset-based pagination is useful when you need to know the total number of pages available. It can also easily support *bi-directional* pagination. Bi-directional pagination allows you to jump back and forth between pages or to navigate to a specific page within the results. This is the kind of navigation typically seen on blogs.

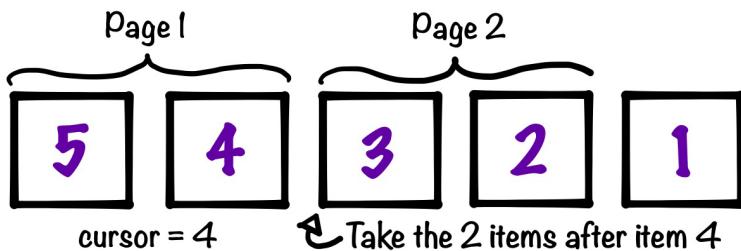
However, there can be performance downsides to this approach if the queried database has a lot of records in it. Further, if new records are added to the database at a high frequency, then the page window may become mismatched with real-time reality, resulting in duplicate or missed records in the pages of results. To illustrate this pitfall, imagine retrieving the first page of results from our dataset. While you're browsing those results, a new sixth item is added before requesting the second page. Suddenly, the paging window shifts back one position, and the fourth item will now confusingly appear at the end of the first page and the start of the second page:



## Cursor-Based

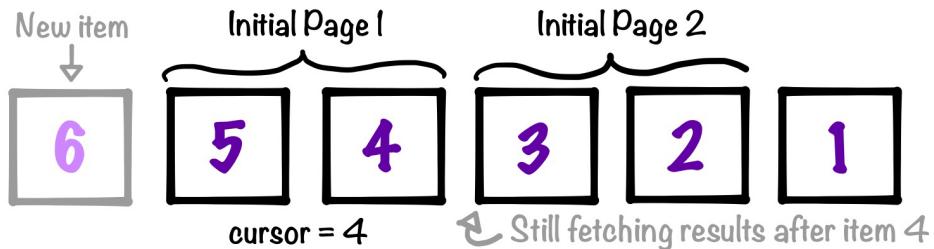
Cursor-based pagination uses (surprise!) a *cursor* to progress through results in a dataset. A cursor is a pointer to a specific result in a dataset and can be anything that makes sense to the back-end application as long as it's a unique, sequential value. As a client navigates through subsequent pages, the server returns results after the item denoted by the cursor value.

Cursor-based pagination on our five-item dataset—once again in descending order with two items per page—can be visualized as follows:



The nature of the cursor itself is inconsequential to the client—the client just needs to send this value back to the server on subsequent requests so the server knows from which point it should retrieve more results.

Cursor-based pagination is well-suited to datasets updated at high velocities because it helps address the issue of page window inaccuracies that can happen with offset-based pagination. If a sixth item is added to our dataset after retrieving the first page, then there will be no confusion about where to start the second page when using a cursor:



This style of pagination does have its trade-offs though. A cursor-based approach has the downside of not providing any way to jump to a specific page number or calculate the total number of pages. However, if you're building an app that will be updated rapidly and with infinite scrolling implemented in the user interface to browse content, then the lack of numbered pages and total page counts likely won't be deal-breakers for you.

## Relay-Style

Relay-style pagination is an opinionated flavor of cursor-based pagination for GraphQL APIs. Relay itself is a JavaScript framework that can be used as a client to retrieve and cache data from a GraphQL API. It was created by Facebook and was designed with Facebook-level applications in mind. In other words, it works well for apps with lots of data in lists that are read and written at a high velocity.

Relay's barriers to entry are a bit higher than other GraphQL client libraries such as Apollo Client, so Relay itself often isn't the first package developers reach for when getting started with GraphQL. However, Relay offers a useful methodology for how to handle paginated data in GraphQL APIs in what it calls a [cursor connection specification](#).

We'll opt for using Relay-style pagination in the Marked GraphQL API because it will be instructive to see how to implement it from scratch and it will also make our API more future-friendly if any clients using Relay wish to make requests to our GraphQL API in the future.

An important thing to keep in mind with Relay-style pagination is that it is *uni-directional* by design. If you need to implement "Previous Page" and "Next Page" buttons to traverse content in an app, then Relay-style pagination probably won't work well for you (although a quick Google search will reveal some proposed workarounds for supporting bi-directional paging with Relay). However, if your user interface requires infinite scrolling to load additional pages of results, then this approach will be a good fit for you.

As we will see, Relay is very opinionated about how pagination requests are made via field arguments as well as how the paginated lists of data are output from the operations. Here's an example of what a query would look like for a single user with this kind of pagination applied to the `network` field:

### GraphQL Query

```
query {
  profile(username: "marksalot") {
    fullName
    network(first: 20, after: "someProfileId") {
      edges {
        cursor
        node {
          fullName
        }
      }
      pageInfo {
        hasPreviousPage
        hasNextPage
      }
    }
  }
}
```

```
    }  
}
```

You may have noticed a few interesting aspects of this query. First, we have the `edges` field, which is a list containing the `edge` type. The `edge` type is an Object type with at least two fields called `node` and `cursor`. The `node` is the Object type itself and can be just about any GraphQL type except a list (for our case, it will be a `Profile` type). The `cursor` is a string that corresponds to the unique, sequential value that identifies the edge. Lastly, the `pageInfo` field returns an Object type that must contain at least `hasPreviousPage` and `hasNextPage` fields. These fields are both non-nullable Boolean types.

Also worth noting are the `first` and `after` arguments for the query—these are the *forward pagination* arguments. If we wanted to paginate backward, then we would use the `last` and `before` arguments.

*Backward pagination* is different from sorting results in descending order. Backward pagination means starting at what constitutes the end of a dataset and working back to the beginning. Sorting results in descending order means traversing pages of results from the item with the highest sort value to the lowest.

A dataset would typically be sorted first, and then would have forward or backward pagination applied to retrieve pages of sorted results. The diagrams in the next section will help to visualize these concepts.

What's not obvious from the example query above is that at the top level a *connection type* would be implemented as an Object type (with the suffix `Connection` added to its name) and the `network` field would now return that single connection Object type instead of a list containing `Profile` types.

The query example above is exactly what we'll be working toward first—we want to be able to paginate the list of profiles that represent members of a user's network. Presumably, a user will connect with many other users over time and if we wanted to list all of the network members in a single view in the client application, then it wouldn't be very efficient to try to load and display them all at once. Pagination will help solve this problem for us here and we'll reuse much of the code we write for this field to paginate the `profiles` and `searchProfiles` queries, as well as a bookmark-related fields in the next chapter.

## Relay-Style Pagination Internals

Relay has a pagination algorithm that describes how to evaluate pagination-related query arguments to return the correct edges in a response. Before implementing this algorithm in our code, we're going to further explore the different elements of Relay-style pagination and

how they work under the hood. First, when applying pagination the client is required to send us information describing how many results it wants, and potentially, where to begin extracting those results from the dataset.

Forward pagination arguments include:

- `first`: The number of results to retrieve, counting forward (must be a non-negative integer)
- `after`: Start counting results after the item identified by this cursor (for example, a database ID)

Backward pagination arguments include:

- `last`: The number of results to retrieve, counting backward (also a non-negative integer)
- `before`: Start counting results before the item identified by this cursor (again, some meaningful, sequential identifier)

Typically, we wouldn't want a client to supply both `first` and `last` arguments. According to the specification:

When both `first` and `last` are included, both of the fields should be set according to the above algorithms, but their meaning as it relates to pagination becomes unclear. This is among the reasons that pagination with both `first` and `last` is discouraged.

Supplying both arguments could certainly be confusing. For example, what does it mean to simultaneously retrieve the first two and last three items from a dataset and within a single page window? To avoid this ambiguity, we're going to throw an error if a client supplies both `first` and `last` arguments.

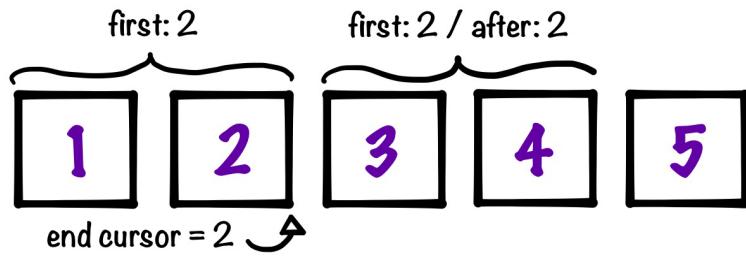
When an initial page of results is sent, the server enables the client to continue requesting additional pages of results by including the following information in the response:

- Whether a next page of results is available (required for forward pagination, optional for backward pagination if it can be efficiently calculated)
- Whether a previous page of results is available (required for backward pagination, optional for forward pagination if it can be efficiently calculated)
- The start cursor and end cursor values for this page (not required in the specification, but useful for clients to formulate their subsequent requests)

With these points in mind, let's now visualize how forward and backward pagination will work on our fictional dataset of five items with an `items` query, sorted in ascending and descending order while requesting two items per page.

## Forward Pagination with Ascending Sort Order

We'll start at the first item (overall, or after a specified cursor) and show items in ascending order:



Page 1 (first: 2):

*API Response*

```
{  
  "data": {  
    "items": {  
      "edges": [  
        {  
          "node": {  
            "id": 1  
          }  
        },  
        {  
          "node": {  
            "id": 2  
          }  
        }  
      ]  
    }  
  }  
}
```

Page 2 (first: 2 / after: 2):

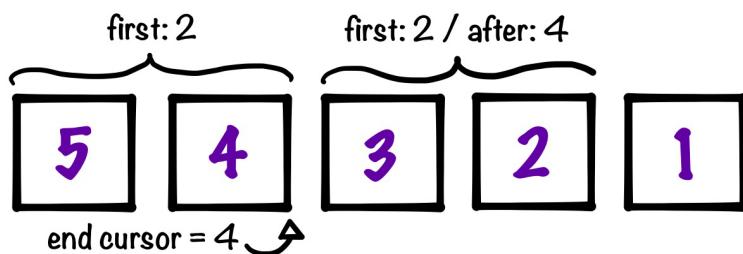
*API Response*

```
{  
  "data": {  
    "items": {  
      "edges": [  
        {  
          "node": {  
            "id": 3  
          }  
        }  
      ]  
    }  
  }  
}
```

```
        }
    },
{
  "node": {
    "id": 4
  }
}
]
}
}
```

## Forward Pagination with Descending Sort Order

We'll start at the last item (overall, or after a specified cursor) and show items in descending order:



Page 1 (first: 2):

*API Response*

```
{
  "data": {
    "items": {
      "edges": [
        {
          "node": {
            "id": 5
          }
        },
        {
          "node": {
            "id": 4
          }
        }
      ]
    }
  }
}
```

```
        }
      ]
    }
}
```

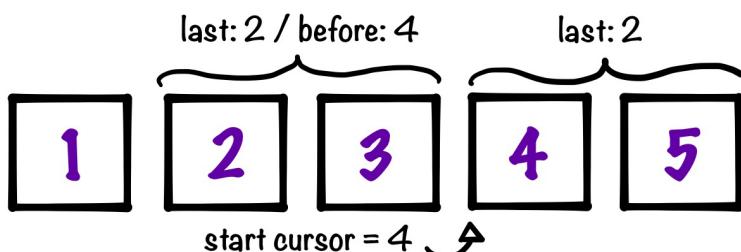
Page 2 (first: 2 / after: 4):

*API Response*

```
{
  "data": {
    "items": {
      "edges": [
        {
          "node": {
            "id": 3
          }
        },
        {
          "node": {
            "id": 2
          }
        }
      ]
    }
  }
}
```

## Backward Pagination with Ascending Sort Order

We'll start at the last item (overall, or before a specified cursor) and show the items in ascending order:



Page 1 (last: 2):

*API Response*

```
{  
  "data": {  
    "items": {  
      "edges": [  
        {  
          "node": {  
            "id": 4  
          }  
        },  
        {  
          "node": {  
            "id": 5  
          }  
        }  
      ]  
    }  
  }  
}
```

Page 2 (last: 2 / before: 4):

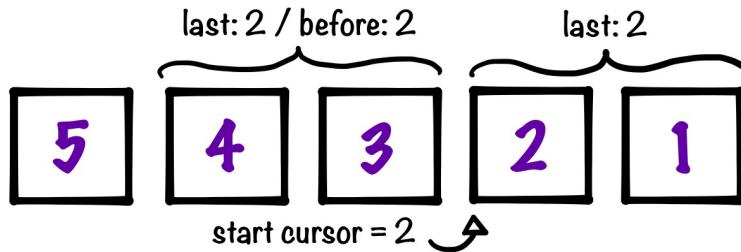
*API Response*

```
{  
  "data": {  
    "items": {  
      "edges": [  
        {  
          "node": {  
            "id": 2  
          }  
        },  
        {  
          "node": {  
            "id": 3  
          }  
        }  
      ]  
    }  
  }  
}
```

```
    }  
}
```

### Backward Pagination with Descending Sort Order

We'll start at the first item (overall, or before a specified cursor) and show items in descending order:



Page 1 (last: 2):

*API Response*

```
{  
  "data": {  
    "items": {  
      "edges": [  
        {  
          "node": {  
            "id": 2  
          }  
        },  
        {  
          "node": {  
            "id": 1  
          }  
        }  
      ]  
    }  
  }  
}
```

Page 2 (last: 2 / before: 2):

*API Response*

```
{  
  "data": {  
    "items": {  
      "edges": [  
        {  
          "node": {  
            "id": 4  
          }  
        },  
        {  
          "node": {  
            "id": 3  
          }  
        }  
      ]  
    }  
  }  
}
```

## Add Pagination Types to the Profiles Service

Now that we have a mental model for how results should be returned in a paginated API response, we'll begin refactoring part of the profiles service's schema to add the new types required by Relay-style pagination. These new types include:

- **ProfileConnection**: An Object type that will be returned for fields that require a paginated list of **Profile** types (contained within its `edges` field).
- **ProfileEdge**: An Object type that will contain `cursor` and `node` fields (and the `node` field will be a single **Profile**).
- **PageInfo**: An Object type containing the `hasPreviousPage` and `hasNextPage` fields required by Relay, as well as the `endCursor` and `startCursor` fields we'll add for a client's convenience (this type will be used across multiple subgraph schemas).

As we refactor our code we'll also add a field argument for sorting the list of user profiles. To begin, we'll create the new **PageInfo** type in the profiles service's schema:

*profiles/src/graphql/schema.graphql*

```
# ...
```

```
"""
Information about pagination in a connection.

"""
type PageInfo {
    "The cursor to continue from when paginating forward."
    endCursor: String
    "Whether there are more items when paginating forward."
    hasNextPage: Boolean!
    "Whether there are more items when paginating backward."
    hasPreviousPage: Boolean!
    "The cursor to continue from them paginating backward."
    startCursor: String
}

# ...
```

Again, the `endCursor` and `startCursor` fields aren't required in the `PageInfo` type in the Relay specification, but providing them here will be a nice convenience for clients so that they don't have to calculate these document ID values manually based on the edges that are returned.

Before we move on, we should pause and think ahead to what the full API schema will look like after the bookmarks service is added in the next chapter. Ultimately, there will be bookmark-related fields that also require pagination so we will need to use the `PageInfo` type in that subgraph schema as well. That brings us to one significant difference between Federation 1 and Federation 2. In Apollo Federation lingo, a non-entity type that is shared among multiple subgraph schemas is known as a *value type*. In Federation 1, these non-entity types (which could be `Scalar`, `Object`, `Interface`, `Input Object`, `Enum`, and `Union` types) were sharable by default as long as they contained exactly the same fields. Although you may not have realized it at the time, we already made use of a value type in the accounts and profiles services by adding the custom `DateTime` `Scalar` to both.

With Federation 2, however, Object types cannot be shared by default. Instead, if you need to share an Object type between subgraphs, then you must import Federation 2's `@shareable` directive and explicitly mark that type (or a sub-selection of its fields) as shareable. Let's do that for the entire `PageInfo` type:

```
profiles/src/graphql/schema.graphql

extend schema
  @link(url: "https://specs.apollo.dev/federation/v2.0",
        import: ["@key", "@shareable"])

# ...
```

```
"""
Information about pagination in a connection.
"""

type PageInfo @shareable {
    # ...
}

# ...
```

This opt-in approach to sharing helps ensure that a subgraph that initially defines an Object type won't find that this type has been used in unexpected ways in other subgraphs. It's important to note that there's nothing magical about the `@shareable` directive that will guarantee that the fields of this type are resolved consistently across all subgraphs that use it—as we discussed with the `DateTime` Scalar type, it's up to the subgraph owners to ensure the field resolvers behave predictably across subgraphs.

### More on Improved Value Type Ergonomics in Federation 2

Beyond the `@shareable` directive, Federation 2 also allows Object types of the same name to differ in what fields they include and also differ in those fields' nullability. This feature of the specification makes it easier to evolve these types if they are shared by many subgraphs and without necessitating a lockstep release for composition to succeed. Additionally, the `@inaccessible` can be used on new fields that are introduced in a shared Object type in one subgraph to exclude these fields from the API schema until other subgraphs provide support for them too.

Enum, Interface, and Union types can also be used more flexibly with Federation 2 as their definitions may differ between subgraphs (and no `@shareable` directive is required). Instead, Enum type values, Union type members, and Interface type fields will be merged during composition.

Please refer to the Apollo Federation documentation to learn more about [the current nuances of value type usage](#).

Next, we'll add the `ProfileConnection` and `ProfileEdge` types:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
```

```
A list of profile edges with pagination information.  
"""  
type ProfileConnection {  
  "A list of profile edges."  
  edges: [ProfileEdge]  
  "Information to assist with pagination."  
  pageInfo: PageInfo!  
}  
  
"""  
A single profile node with its cursor.  
"""  
type ProfileEdge {  
  "A cursor for use in pagination."  
  cursor: ID!  
  "A profile at the end of an edge."  
  node: Profile!  
}  
  
# ...
```

We'll need to add one more thing to the schema to sort the network profiles in ascending or descending order via a field argument. To facilitate sorting, we'll define our first Enum type called `ProfileOrderBy` to help limit the sorting field to either usernames or creation time only, and either ascending or descending alphabetically:

`profiles/src/graphql/schema.graphql`

```
# ...  
  
"""  
Sorting options for profile connections.  
"""  
enum ProfileOrderBy {  
  "Order profiles ascending by creation time."  
  CREATED_AT_ASC  
  "Order profiles descending by creation time."  
  CREATED_AT_DESC  
  "Order profiles ascending by username."  
  USERNAME_ASC  
  "Order profiles descending by username."  
  USERNAME_DESC
```

```
}
```

```
# ...
```

The first portion of the existing schema that we'll refactor to use the pagination-related types will be the `network` field on the `Profile` type:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
A profile contains metadata about a specific user.
"""

type Profile @key(fields: "id") {
    # ...
    "Other users that have been added to the user's network."
    network (
        first: Int
        after: String
        last: Int
        before: String
        orderBy: ProfileOrderBy = USERNAME_ASC
    ): ProfileConnection
    # ...
}

# ...
```

The `ProfileOrderByInput` is an optional variable. If it's not provided, we'll set the default sort order to be ascending by username. The use of a default value for the `orderBy` argument is noteworthy because it guarantees that a value for this argument will be available in the field resolver while still allowing this field to be nullable. We could have also applied a default value at runtime, but defining the default in the schema makes this behavior more transparent to client developers.

Also note that the type we expect to receive back for this field is now a single `ProfileConnection`, rather than a list of `Profile` types, which means it will no longer be a simple array of profiles, but rather an object with the following shape:

```
{
```

```
    "data": {
```

```
        "profile": {
```

```
"username": "marksalot",
"network": {
  "edges": [
    {
      "node": {
        "username": "coolUserInMyNetwork"
      },
      "cursor": "5d06de0941b23b1f0ccc7911"
    }
  ],
  "pageInfo": {
    "startCursor": "5d06de0941b23b1f0ccc7911",
    "endCursor": "5d06de0941b23b1f0ccc7911",
    "hasNextPage": true,
    "hasPreviousPage": false
  }
}
}
```

Before we move on to update the resolver for the `network` field to return data in this shape, we need to figure out how to get paginated results like this from the database. First, we'll write some helper functions to facilitate paginating results from MongoDB. Following that, we'll need to update the `ProfilesDataSource` to use these helpers so that our resolvers can return the correct data in the right shape.

## Create a Pagination Class for MongoDB Data

Creating the functions that will facilitate paginating results from MongoDB will be one of the logically denser tasks we undertake while building this API. The payoff will be worthwhile—by the end of this chapter, we'll have a working implementation of Relay-style pagination to use in other Marked subgraph services and also as a reference point for future projects.

The code we write to paginate the profiles will be generic enough that we can repurpose it in the bookmarks service later, so we'll create a new file called `Pagination.js` in the `shared/src/utils` directory to contain the code for the `Pagination` class. In this class, we will define generalized methods that MongoDB-based data sources can use to retrieve pages of documents by a cursor. Let's scaffold all the methods we'll need here now:

*shared/src/utils/Pagination.js*

```
class Pagination {
  constructor(Model) {
    this.Model = Model;
  }

  // Get documents and cast them into the correct edge/node shape
  async getEdges(queryArgs) {}

  // Get pagination information
  async getPageInfo(edges, queryArgs) {}

  // Add the cursor ID with the correct comparison operator to the query
  // filter
  async _getFilterWithCursor(fromCursorId, filter, operator, sort) {}

  // Create the aggregation pipeline to paginate a full-text search
  async _getSearchPipeline(fromCursorId, filter, first, operator, sort) {}

  // Reverse the sort direction when queries need to look in the opposite
  // direction of the set sort order (e.g. next/previous page checks)
  _reverseSortDirection(sort) {}

  // Get the correct comparison operator based on the sort order
  _getOperator(sort, options = {}) {}

  // Determine if a query is a full-text search based on the sort
  // expression
  _isSearchQuery(sort) {}

  // Check if a next page of results is available
  async _getHasNextPage(endCursor, filter, sort) {}

  // Check if a previous page of results is available
  async _getHasPreviousPage(startCursor, filter, sort) {}

  // Get the ID of the first document in the paging window
  _getStartCursor(edges) {}

  // Get the ID of the last document in the paging window
  _getEndCursor(edges) {}}
```

```
}

export default Pagination;
```

Again, we want this class to be repurposed, so we'll pass a Mongoose model into the constructor when we instantiate a new `Pagination` object. The first two methods—`getEdges` and `getPages`—will be what an Apollo data source uses to fetch the required documents and page info for a given resolver. All of the other methods will be used internally by the class to support the work done in the first two methods.

There's a lot happening in this file already, so let's outline a high-level plan. First, we'll work out how to retrieve edges and page info for forward pagination scenarios (where `first` is supplied as an argument). Next, we'll update the `ProfilesDataSource` and resolvers to test out that code, and then we'll deal with backward pagination scenarios (where `last` is supplied as an argument) and test those too. To begin, we'll import `UserInputError` from `apollo-server` at the top of `Pagination.js` to help with error handling:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";
// ...
```

Next, we'll plan out the `getEdges` method:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getEdges(queryArgs) {
    const {
      after,
      before,
      first,
      last,
      filter = {},
      sort = {}
    } = queryArgs;
    let edges;
```

```
// handle user input errors...

if (first) {
  // forward pagination happens here
  // edges = ???
} else {
  // backward pagination happens here (later)
  // edges = ???
}

  return edges;
}

// ...
}

export default Pagination;
```

This method has one parameter called `queryArgs`. The `queryArgs` will be an object containing the Relay-related pagination arguments, as well as inner `filter` and `sort` objects to help MongoDB pull the correct, ordered documents from the database. The method is set up with basic control flow to check if the query is requesting forward-paginated or backward-paginated documents. Before we attempt to pull any paginated results, we'll do some error handling:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getEdges(queryArgs) {
    const { after, before, first, last, filter = {}, sort = {} } =
      queryArgs;
    let edges;

    if (!first && !last) {
      throw new UserInputError(
        "Provide a `first` or `last` value to paginate this connection."
      );
    } else if (first && last) {
      throw new UserInputError(
```

```
        "Passing `first` and `last` arguments is not supported with this
        connection."
    );
} else if (first < 0 || last < 0) {
    throw new UserInputError(
        "Minimum record request for `first` and `last` arguments is 0."
    );
} else if (first > 100 || last > 100) {
    throw new UserInputError(
        "Maximum record request for `first` and `last` arguments is 100."
    );
}

// ...
}

// ...
}

export default Pagination;
```

The `first` or `last` argument is key to understanding how the profiles must be paginated from the database, and we need one (and only one) of those arguments to get the correct documents from MongoDB. Correspondingly, we'll throw an error if the client doesn't supply a `first` or `last` argument, or if it supplies both `first` and `last` arguments. According to the Relay specification, we must also throw an error if the `first` or `last` argument supplied is less than `0`. For performance reasons we'll also limit the maximum number of results a client can request at one time to `100`.

The Relay Specification doesn't say that supplying both `first` and `last` arguments is forbidden, but it does point out (twice) that this practice is strongly discouraged because it leads to ambiguous pagination query results. We'll choose to throw an error if this occurs to avoid confusion.

Next, we'll fetch documents when the `first` argument is available:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
```

```
// ...  
  
async getEdges(queryArgs) {  
    // ...  
  
    if (first) {  
        const docs = await this.Model.find(filter)  
            .sort(sort)  
            .limit(first)  
            .exec();  
  
        edges = docs.length  
            ? docs.map(doc => ({ cursor: doc._id, node: doc }))  
            : [];  
    } else {  
        // backward pagination happens here (later)  
        // edges = ???  
    }  
}  
  
// ...  
}  
  
export default Pagination;
```

Note that we pass the `filter` object into the `find` method above to filter the selection of returned documents. Recall that we filter the `network` field on the `Profile` type by searching for applicable user account IDs as follows:

```
{ accountId: { $in: [ "_id_1", "_id_2", ... ] } }
```

By using the `$in` operator here we are asking MongoDB to only return documents where the `accountId` field matches one of the account IDs in the provided array, which is exactly what we need—only return documents where the `accountId` matches one of the account IDs in the `network` array field of a given user profile document.

We also chain the `sort` method onto this and pass in the `sort` object, which by default will be `{username: 1}`, meaning that we want to sort by `username` in ascending order as a default. To sort by `username` in descending order, we would set `{username: -1}`. Next, we chain the `limit` method to only return the first N results, as specified by the `first` value. After retrieving the documents from MongoDB, we must `map` over the array of documents to produce a new array of objects with `cursor` and `node` properties so that they match the `ProfileEdge` shape.

At first glance, this may seem like all we need. If we only ever needed to retrieve the first page of results, then that would be the case. However, we must account for a client supplying the `after` argument for a paginated field. To do that we have to build out the `_getFilterWithCursor` method. This method will allow us to take the filter that is passed into `getEdges` via its `filter` parameter and combine it with another filter that only queries results after a specific document. The query document containing the combined filters will have this shape:

```
{ $and: [
  { { accountId: { $in: [ "_id_1", "_id_2", ... ] } } },
  { username: { $gt: "someuser" } }
]}
```

The `$gt` operator means *greater than* and will only get results *after* the document that contains the specified username. Later, we'll also use the operator for *less than*, `$lt`, to look for results that occur *before* a specific username.

The `_getFilterWithCursor` method will look like this:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async _getFilterWithCursor(fromCursorId, filter, operator, sort) {
    let filterWithCursor = { $and: [filter] };
    const fieldArr = Object.keys(sort);
    const field = fieldArr.length ? fieldArr[0] : "_id";
    const fromDoc = await this.Model.findOne({ _id: fromCursorId })
      .select(field)
      .exec();

    if (!fromDoc) {
      throw new UserInputError(`No record found for ID '${fromCursorId}'`);
    }

    filterWithCursor.$and.push({
      [field]: { [operator]: fromDoc[field] }
    });

    return filterWithCursor;
  }
}
```

```
// ...
}

export default Pagination;
```

Note that this method has four parameters:

- `fromCursorId`: Typically, this will be either the `after` or `before` value
- `filter`: An array containing the followed profile IDs
- `sort`: The MongoDB sort object
- `operator`: Specifies whether we want results `$lt` or `$gt` than the provided cursor ID

Inside the method, we nest the original `filter` inside of an array within another object containing MongoDB's `$and` logical operator as a key. We'll push our cursor rule onto this array shortly. Next, we check if the `sort` object is not empty, and assign its single key (the `username` here) to the `field` variable. If it's empty we just use the `_id` field by default.

After that, we need to get the document that matches `fromCursorId` on the `_id` field. If one can't be found, then we throw an error because we won't have anything to paginate against. If everything is OK, we push the additional filter onto the `$and` array and return the updated filter from the function. Note that we also chain on the `select` method before `exec` and pass it the `field` value as an argument because we don't need to retrieve all of the fields in the `fromDoc`, just the one we use for sorting (the `_id` field will also be included by default unless we explicitly opt out of retrieving it). This is known as a *projection* in MongoDB.

Next, we have to build out the `_getOperator` method to get the correct operator to pass into `_getFilterWithCursor`:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
    // ...

    _getOperator(sort, options = {}) {
        const orderArr = Object.values(sort);
        return orderArr.length && orderArr[0] === -1 ? "$lt" : "$gt";
    }

    // ...
}
```

```
export default Pagination;
```

For now, what happens in this method will be straightforward—it checks if there are any values inside of the `sort` object, and if the value is `-1` the returned comparison operator is `$lt`, otherwise, it's `$gt`. Later, we'll need to create more advanced control flow here and make use of the `options` parameter when we add paginated results for full-text search queries.

Now we can update the `getEdges` method to handle an `after` argument to ensure we get the correct page of results after a cursor:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getEdges(queryArgs) {
    // ...

    if (first) {
      const operator = this._getOperator(sort);
      const queryDoc = after
        ? await this._getFilterWithCursor(after, filter, operator, sort)
        : filter;

      const docs = await this.Model.find(queryDoc)
        .sort(sort)
        .limit(first)
        .exec();

      edges = docs.length
        ? docs.map(doc => ({ cursor: doc._id, node: doc }))
        : [];
    } else {
      // backward pagination happens here (later)
      // edges = ???
    }
  }

  // ...
}
```

```
export default Pagination;
```

In the code above, we generate the combined query document if `after` has a value, otherwise we'll stick with the regular `filter` as a query document to get the first page of results. We also get the correct operator so when we pass in the new `queryDoc` value to the `find` method we will get results in the correct sort direction.

Before we move onto updating the network resolver and its related `ProfilesDataSource` method, we'll set up the `getPageInfo` method to return some placeholder values for now:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getPageInfo(edges, queryArgs) {
    return {
      hasNextPage: false,
      hasPreviousPage: false,
      startCursor: null,
      endCursor: null
    };
  }

  // ...
}

export default Pagination;
```

We'll fix this later when we get basic forward and backward pagination working. And as a final step, we'll export the new `Pagination` class from the main entry point of the shared module:

`shared/src/index.js`

```
import authDirectives from "./directives/authDirectives.js";
import DateTimeType from "./scalars/DateTimeType.js";
import Pagination from "./utils/Pagination.js";

export { authDirectives, DateTimeType, Pagination };
```

## Add Forward Pagination to the `network` Field

Before moving forward, you'll want to make sure that you have at least six user accounts and associated profiles created in Auth0 and MongoDB. Use the `createAccount` and `createProfile` mutations to create a few additional users if you haven't done so already.

To speed things up, you may want to temporarily remove the `@owner` directive for `createProfile` so that you don't have to obtain an access token for each user account to create their profile. But be sure to reapply the directive when you're done!

Once the users are created, make sure that one of your users has the other five users added to their network. You can use the `addToNetwork` query in Explorer to do this. Because we can't use the API to query data on the `network` field until we finish refactoring its resolver in this section, you'll have to check in MongoDB to make sure that you correctly followed the five other users. You can see five users added to the `network` field in first profile document in MongoDB Compass below:

```

_id: ObjectId("62c6523f6aa66d4dfdd2563")
accountId: "auth0|625b5a7847a7f7006f3ce7ab"
fullName: "Sir Marks-a-Lot"
> interests: Array
> network: Array
  0: "auth0|625d0d3935940b000f4eb638"
  1: "auth0|625d0d591109db000fa6e67d8"
  2: "auth0|625de0d69e9b3674a00701f91d1"
  3: "auth0|625de0d798e5ab40068ace5cae"
  4: "auth0|625de0d83993674a00701f91da"
username: "marksalot"
createdAt: 2022-04-17 13:00:11.511
__v: 0

_id: ObjectId("62c6580f6aa66d4dfdd2565")
accountId: "auth0|625c652808eac7006851b39f"
fullName: "Marky Marked"
> interests: Array
> network: Array
username: "funkybunch"
createdAt: 2022-04-17 13:07:44.718
__v: 0
    
```

For reference, the usernames, Auth0 IDs, and profile document IDs of the five network members referenced in the examples throughout this chapter are:

```
ada_lovelace|auth0|625ed039359d0b006f4eb638|625ed5a652707218760b0122  
alan_turing|auth0|625ed0591109db006a6e67d8|625ed61552707218760b0124  
george_boole|auth0|625ed06e9b367400701f91d1|625ed64852707218760b0126  
grace_hopper|auth0|625ed0798e5ab40068ace5ae|625ed69a52707218760b0128  
margaret_hamilton|auth0|625ed0839b367400701f91da|625ed6c052707218760b012a
```

The order of the users listed above reflects the order in which they were added to MongoDB.

With our `Pagination` class sufficiently built out, we'll import it into `ProfilesDataSource.js`:

```
profiles/src/graphql/dataSources/ProfilesDataSource.js
```

```
import { DataSource } from "apollo-datasource";  
import { UserInputError } from "apollo-server";  
  
import { Pagination } from "../../../../../shared/src/index.js";  
  
// ...
```

We'll use it to create a new `Pagination` object, passing the `Profile` model into it:

```
profiles/src/graphql/dataSources/ProfilesDataSource.js
```

```
// ...  
  
class ProfilesDataSource extends DataSource {  
  constructor({ Profile }) {  
    super();  
    this.Profile = Profile;  
    this.pagination = new Pagination(Profile);  
  }  
  
  // ...  
}  
  
export default ProfilesDataSource;
```

Now we can update the `getNetworkProfiles` method:

*profiles/src/graphql/dataSources/ProfilesDataSource.js*

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    async getNetworkProfiles({ after, before, first, last, orderBy, network }) {
        let sort = {};
        const sortArgs = orderBy.split("_");
        const direction = sortArgs.pop();
        const field = sortArgs
            .map(arg => arg.toLowerCase())
            .map((arg, i) =>
                i === 0 ? arg : arg.charAt(0).toUpperCase() + arg.slice(1)
            )
            .join("");
        sort[field] = direction === "DESC" ? -1 : 1;

        const filter = { accountId: { $in: network } };
        const queryArgs = { after, before, first, last, filter, sort };
        const edges = await this.pagination.getEdges(queryArgs);
        const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

        return { edges, pageInfo };
    }

    // ...
}

export default ProfilesDataSource;
```

Instead of only passing in the `network` array to this method, we now pass in an object containing all of the information that we'll need to make our paginated query. Next, instead of passing `{ accountId: { $in: network } }` directly into the model's `find` method, we create a `sort` object based on the `ProfileOrderBy` Enum that was passed as an argument to the field. We split the `Enum` value at its underscore characters and pop off the `sort` direction at the end, then convert the remaining items back into camel case string. We set the direction of the `sort` field to `-1` or `1` based on whether the `Enum` value indicated descending or ascending direction. And we assume that an `orderBy` value will always be available because we set the default value for this `field` argument to be `USERNAME_ASC`.

After that, we organize our query arguments into one `queryArgs` object and pass it into the `Pagination` object's `getEdges` method. We then use the resulting `edges` as an argument to the `getPageInfo` method with the `queryArgs`. Finally, we return the `edges` and the `pageInfo` objects in the shape that the `network` field resolver now expects.

Thinking ahead to where else we may use pagination in this service's schema, it seems like it would be a good idea to abstract away the code that sets the sort field and direction into a separate method. To that end, we'll create a `_getProfileSort` method:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    _getProfileSort(sortEnum) {
        let sort = {};
        const sortArgs = sortEnum.split("_");
        const direction = sortArgs.pop();
        const field = sortArgs
            .map(arg => arg.toLowerCase())
            .map((arg, i) =>
                i === 0 ? arg : arg.charAt(0).toUpperCase() + arg.slice(1)
            )
            .join("");
        sort[field] = direction === "DESC" ? -1 : 1;

        return sort;
    }

    // ...
}

export default ProfilesDataSource;
```

Let's update the `getNetworkProfiles` method to use the `_getProfileSort` helper:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
```

```
// ...

async getNetworkProfiles({ after, before, first, last, orderBy, network }) {
  const sort = this._getProfileSort(orderBy);
  const filter = { accountId: { $in: network } };
  const queryArgs = { after, before, first, last, filter, sort };
  const edges = await this.pagination.getEdges(queryArgs);
  const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

  return { edges, pageInfo };
}

// ...
}

export default ProfilesDataSource;
```

We can now update the `network` field resolver in `resolvers.js`:

`profiles/src/graphql/resolvers.js`

```
import { UserInputError } from "apollo-server";

const resolvers = {
  // ...

  Profile: {
    // ...
    network(profile, args, { dataSources }) {
      return dataSources.profilesAPI.getNetworkProfiles({
        ...args,
        network: profile.network
      });
    },
    // ...
  },
  // ...
};

export default resolvers;
```

In the updated resolver above, instead of only passing the array of network member account IDs to the data source method we must pass in an object containing both this array and the pagination arguments that will be contained in `args`.

Now we're ready to test the query:

#### GraphQL Query

```
query Network($username: String!, $first: Int) {
  profile(username: $username) {
    username
    network(first: $first) {
      edges {
        cursor
        node {
          username
        }
      }
      pageInfo {
        endCursor
        hasNextPage
        hasPreviousPage
        startCursor
      }
    }
  }
}
```

#### Query Variables

```
{
  "username": "marksalot",
  "first": 2
}
```

#### API Response

```
{
  "data": {
    "profile": {
      "username": "marksalot",
      "network": {
        "edges": [
```

```
{  
  "cursor": "625ed5a652707218760b0122",  
  "node": {  
    "username": "ada_lovelace"  
  }  
},  
{  
  "cursor": "625ed61552707218760b0124",  
  "node": {  
    "username": "alan_turing"  
  }  
}  
],  
"pageInfo": {  
  "endCursor": null,  
  "hasNextPage": false,  
  "hasPreviousPage": false,  
  "startCursor": null  
}  
}  
}  
}  
}
```

As expected, we get back the first two results sorted by username in ascending alphabetical order. We can also see that pageInfo returns the placeholder values. Try adding an orderBy argument now to the query to ensure it works as expected:

#### *GraphQL Query*

```
query Network($username: String!, $first: Int, $orderBy: ProfileOrderBy) {  
  profile(username: $username) {  
    username  
    network(first: $first, orderBy: $orderBy) {  
      edges {  
        cursor  
        node {  
          username  
        }  
      }  
    }  
  }  
}
```

```
    }  
}
```

### Query Variables

```
{  
  "username": "marksalot",  
  "first": 2,  
  "orderBy": "USERNAME_DESC"  
}
```

### API Response

```
{  
  "data": {  
    "profile": {  
      "username": "marksalot",  
      "network": {  
        "edges": [  
          {  
            "cursor": "625ed6c052707218760b012a",  
            "node": {  
              "username": "margaret_hamilton"  
            }  
          },  
          {  
            "cursor": "625ed69a52707218760b0128",  
            "node": {  
              "username": "grace_hopper"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

Now we see the last two profiles on the first page of results because they are sorted in descending alphabetical order. As a final test of forward pagination, try querying for the second page of results using the `after` argument. We'll use a specific user's profile ID (the example below uses Alan Turing's ID) so we can see what profiles come after him in ascending order:

*GraphQL Query*

```
query Network($username: String!, $first: Int, $after: String) {
  profile(username: $username) {
    username
    network(first: $first, after: $after) {
      edges {
        cursor
        node {
          username
        }
      }
    }
  }
}
```

*Query Variables*

```
{
  "username": "marksalot",
  "first": 2,
  "after": "625ed61552707218760b0124"
}
```

*API Response*

```
{
  "data": {
    "profile": {
      "username": "marksalot",
      "network": {
        "edges": [
          {
            "cursor": "625ed64852707218760b0126",
            "node": {
              "username": "george_boole"
            }
          },
          {
            "cursor": "625ed69a52707218760b0128",
            "node": {
              "username": "grace_hopper"
            }
          }
        ]
      }
    }
  }
}
```

```
        }
      ]
    }
}
```

## Add Backward Pagination to the `network` Field

Forward pagination is functional, so we'll finish building out backward pagination now. We first need to complete the `_reverseSortDirection` method in `Pagination.js`. This method will flip the sort direction when using backward pagination so that we can work from the end of the results and provide what would ordinarily be the end page at the beginning. To do this, we must set the sort object's key to be `-1` if it's initially `1` and `1` if it's initially `-1`:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  _reverseSortDirection(sort) {
    const fieldArr = Object.keys(sort);

    if (fieldArr.length === 0) {
      return { $natural: -1 };
    }

    const field = fieldArr[0];
    return { [field]: sort[field] * -1 };
  }

  // ...
}

export default Pagination;
```

Inside this method, we check if the sort object was empty (indicated by the array resulting from `Object.keys` having a length of `0`). If nothing was provided as a sort field, then we flip the

\$natural sort order, which is the default sort order that MongoDB refers to the documents on disk. We return a new object with the reserved sort order from this method because we don't want to mutate the original sort object (doing so would create bugs elsewhere in the pagination code).

Now we'll put `_reverseSortDirection` to use in the `getEdges` method so it supports backward pagination when a `last` argument is provided:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getEdges(queryArgs) {
    // ...

    if (first) {
      // ...
    } else {
      const reverseSort = this._reverseSortDirection(sort);
      const operator = this._getOperator(reverseSort);

      const queryDoc = before
        ? await this._getFilterWithCursor(before, filter, operator,
          reverseSort)
        : filter;

      const docs = await this.Model.find(queryDoc)
        .sort(reverseSort)
        .limit(last)
        .exec();

      edges = docs.length
        ? docs.map(doc => ({ node: doc, cursor: doc._id })).reverse()
        : [];
    }
  }

  export default Pagination;
```

This code is very similar to forward pagination, but with three notable differences. First, we call the `_reverseSortDirection` method, passing in the `sort` object from the `queryArgs`, and set the `reverseSort` variable based on the return value. We reverse the sort order internally (for example, `{ username: 1 }` becomes `{ username: -1 }`) because with backward pagination, even if we want the individual page of the results to be in ascending order, we still need MongoDB to start its query from the end of the results and then move toward the front from there. The reverse would be true for backward pagination in descending order.

The second difference is that the ternary expression that sets the `queryDoc` value checks `before` instead of `after`. Finally, we chain the `reverse` array method onto the output of `map` because even though we have to start counting documents from the end (this is why we flipped the sort direction at the beginning), we still want the individual results on each page to be in the correct ascending or descending order, so `reverse` puts them back in the expected order.

Now we can restart the profiles service to pick up the changes made to the `Pagination` class and test out backward pagination:

#### GraphQL Query

```
query Network($username: String!, $last: Int) {
  profile(username: $username) {
    username
    network(last: $last) {
      edges {
        cursor
        node {
          username
        }
      }
    }
  }
}
```

#### Query Variables

```
{
  "username": "marksalot",
  "last": 2
}
```

#### API Response

```
{
  "data": {
```

```
"profile": {
  "username": "marksalot",
  "network": {
    "edges": [
      {
        "cursor": "625ed69a52707218760b0128",
        "node": {
          "username": "grace_hopper"
        }
      },
      {
        "cursor": "625ed6c052707218760b012a",
        "node": {
          "username": "margaret_hamilton"
        }
      }
    ]
  }
}
```

We can get the next (but really previous!) page of results using Grace Hopper's document ID:

#### *GraphQL Query*

```
query Network($username: String!, $last: Int, $before: String) {
  profile(username: $username) {
    username
    network(last: $last, before: $before) {
      edges {
        cursor
        node {
          username
        }
      }
    }
  }
}
```

*Query Variables*

```
{  
  "username": "marksalot",  
  "last": 2,  
  "before": "625ed69a52707218760b0128",  
}
```

*API Response*

```
{  
  "data": {  
    "profile": {  
      "username": "marksalot",  
      "network": {  
        "edges": [  
          {  
            "cursor": "625ed61552707218760b0124",  
            "node": {  
              "username": "alan_turing"  
            }  
          },  
          {  
            "cursor": "625ed64852707218760b0126",  
            "node": {  
              "username": "george_boole"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

Now try out descending order with backward pagination:

*GraphQL Query*

```
query Network($username: String!, $last: Int, $orderBy: ProfileOrderBy) {  
  profile(username: $username) {  
    username  
    network(last: $last, orderBy: $orderBy) {  
      edges {
```

```
        cursor
      node {
        username
      }
    }
  }
}
```

*Query Variables*

```
{
  "username": "marksalot",
  "last": 2,
  "orderBy": "USERNAME_DESC"
}
```

*API Response*

```
{
  "data": {
    "profile": {
      "username": "marksalot",
      "network": {
        "edges": [
          {
            "cursor": "625ed61552707218760b0124",
            "node": {
              "username": "alan_turing"
            }
          },
          {
            "cursor": "625ed5a652707218760b0122",
            "node": {
              "username": "ada_lovelace"
            }
          }
        ]
      }
    }
  }
}
```

## Generate the PageInfo to Include in the Response

With forward and backward pagination functional, we can put the final pieces of the pagination puzzle in place by resolving real data for `PageInfo`. Our first step will be to write the `_getStartCursor` and `_getEndCursor` methods. Respectively, these methods only need to return the first and last cursor IDs from a list of queried edges:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  _getStartCursor(edges) {
    if (!edges.length) {
      return null;
    }

    return edges[0].cursor;
  }

  _getEndCursor(edges) {
    if (!edges.length) {
      return null;
    }

    return edges[edges.length - 1].cursor;
  }
}

export default Pagination;
```

Now we'll work on the `_getHasNextPage` method. This method's job is to determine if there's at least one document beyond the current page, and then return `true` if a document exists or `false` if one doesn't. The Relay specification indicates that we don't necessarily need to provide a value for `hasNextPage` when paginating backward—this is only recommended when it is efficient to do so. Otherwise, we could just return `false` in the backward pagination scenario. It's reasonably efficient for us to calculate this so we will be able to supply a real value here. The same will be true for forward pagination when checking for a previous page later on:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async _getHasNextPage(endCursor, filter, sort) {
    const operator = this._getOperator(sort);
    const queryDoc = await this._getFilterWithCursor(
      endCursor,
      filter,
      operator,
      sort
    );

    const nextPage = await this.Model.findOne(queryDoc)
      .select("_id")
      .sort(sort);

    return Boolean(nextPage);
  }

  // ...
}

export default Pagination;
```

In this method, we create a `queryDoc` variable like before, but this time we use the `endCursor` value as the `fromCursorId` argument because we want to know if anything exists beyond the end cursor. Next, we pass the `queryDoc` into `findOne` and then chain on `select` to only retrieve the `_id` field for the document because we don't care about the content of the document—we just want to know that it exists and then apply the sort. Lastly, we coerce the returned document or `null` into a boolean so we can return `true` or `false` from this method.

Now we'll create the `_getHasPreviousPage` method in a similar fashion:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...
```

```
async _getHasPreviousPage(startCursor, filter, sort) {
  const reverseSort = this._reverseSortDirection(sort);
  const operator = this._getOperator(reverseSort);
  const queryDoc = await this._getFilterWithCursor(
    startCursor,
    filter,
    operator,
    reverseSort
  );

  const prevPage = await this.Model.findOne(queryDoc)
    .select("_id")
    .sort(reverseSort);

  return Boolean(prevPage);
}

// ...
}

export default Pagination;
```

As we did with backward pagination, we must again reverse the sort direction. To determine whether there's a previous page we need to flip the comparison operator and then check if at least one result comes before the `startCursor` for ascending pagination, or at least one result comes after it for descending pagination.

Finally, we can update `getPageInfo` using all of the new methods to calculate real values if there are any items in the `edges` array:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getPageInfo(edges, queryArgs) {
    if (edges.length) {
      const { filter = {}, sort = {} } = queryArgs;
      const startCursor = this._getStartCursor(edges);
      const endCursor = this._getEndCursor(edges);
```

```
const hasNextPage = await this._getHasNextPage(
  endCursor,
  filter,
  sort
);
const hasPreviousPage = await this._getHasPreviousPage(
  startCursor,
  filter,
  sort
);

return {
  hasNextPage,
  hasPreviousPage,
  startCursor,
  endCursor
};

return {
  hasNextPage: false,
  hasPreviousPage: false,
  startCursor: null,
  endCursor: null
};
}

// ...
}

export default Pagination;
```

If we perform any queries now, we should be able to see accurate pageInfo. Try querying the second page of results and confirm that hasPreviousPage and hasNextPage are both true:

#### *GraphQL Query*

```
query Network($username: String!, $first: Int, $after: String) {
  profile(username: $username) {
    username
    network(first: $first, after: $after) {
      edges {
```

```
        cursor
      node {
        username
      }
    }
  pageInfo {
    hasPreviousPage
    hasNextPage
    startCursor
    endCursor
  }
}
}
```

#### Query Variables

```
{
  "username": "marksalot",
  "first": 2,
  "after": "625ed61552707218760b0124"
}
```

#### API Response

```
{
  "data": {
    "profile": {
      "username": "marksalot",
      "network": {
        "edges": [
          {
            "cursor": "625ed64852707218760b0126",
            "node": {
              "username": "george_boole"
            }
          },
          {
            "cursor": "625ed69a52707218760b0128",
            "node": {
              "username": "grace_hopper"
            }
          }
        ]
      }
    }
  }
}
```

```
        }
    ],
    "pageInfo": {
        "hasPreviousPage": true,
        "hasNextPage": true,
        "startCursor": "625ed64852707218760b0126",
        "endCursor": "625ed69a52707218760b0128"
    }
}
}
```

## Add Pagination to `profiles` and `searchProfiles` Fields

With pagination up and running for the `network` field on the `Profile` type, we'll finish up this chapter by adding pagination for the `profiles` and `searchProfiles` queries. Adding pagination to the `profiles` field will be easier than `searchProfiles`, so we'll start there.

First, we'll update the `profiles` field in the `profiles` service's schema to accept pagination-related arguments. We also need to update it to return a single `ProfileConnection` now instead:

`profiles/src/graphql/schema.graphql`

```
# ...

type Query {
    # ...
    "Retrieves a list of profiles."
    profiles(
        after: String
        before: String
        first: Int
        last: Int
        orderBy: ProfileOrderBy = USERNAME_ASC
    ): ProfileConnection @private
    # ...
}

# ...
```

Next, we'll update the `getProfiles` method in the `ProfilesDataSource` class:

profiles/src/graphql/dataSources/ProfilesDataSource.js

```
// ...

class ProfilesDataSource extends DataSource {
  // ...

  async getProfiles({ after, before, first, last, orderBy }) {
    const sort = this._getProfileSort(orderBy);
    const queryArgs = { after, before, first, last, sort };
    const edges = await this.pagination.getEdges(queryArgs);
    const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

    return { edges, pageInfo };
  }

  // ...
}

export default ProfilesDataSource;
```

The `getProfiles` method must be `async` now because we `await` the promises to get both the `edges` and `pageInfo` before returning an object of the combined data from this function. We must also update the `getProfiles` resolver in `resolvers.js` so that it passes the pagination args through:

profiles/src/graphql/resolvers.js

```
import { UserInputError } from "apollo-server";

const resolvers = {
  // ...

  Query: {
    // ...
    profiles(parent, args, { dataSources }) {
      return dataSources.profilesAPI.getProfiles(args);
    },
    // ...
  },
  // ...
};
```

```
export default resolvers;
```

The paginated `profiles` query is ready for testing:

*GraphQL Query*

```
query Profiles($first: Int) {
  profiles(first: $first) {
    edges {
      node {
        username
      }
    }
    pageInfo {
      hasPreviousPage
      hasNextPage
      startCursor
      endCursor
    }
  }
}
```

*Query Variables*

```
{
  "first": 2
}
```

*API Response*

```
{
  "data": {
    "profiles": {
      "edges": [
        {
          "node": {
            "username": "ada_lovelace"
          }
        },
        {
          "node": {
            "username": "alan_turing"
          }
        }
      ]
    }
  }
}
```

```
        "username": "alan_turing"
    }
}
],
"pageInfo": {
    "hasPreviousPage": false,
    "hasNextPage": true,
    "startCursor": "625ed5a652707218760b0122",
    "endCursor": "625ed61552707218760b0124"
}
}
}
```

Adding pagination to the `searchProfiles` query will take a bit effort to implement. The first step will be to update the `searchProfiles` query definition:

`profiles/src/graphql/schema.graphql`

```
# ...

type Query {
# ...
"Performs a search of user profiles. Results are available in descending
order by relevance only."
searchProfiles(
    after: String
    first: Int
    "The text string to search for in usernames or full names."
    query: String!
): ProfileConnection @private
}

# ...
```

Unlike the `network` and the `profiles` fields, we left out the `before` and `last` arguments that enable backward pagination. That means we're deviating from the Relay pagination specification slightly, but there are two good reasons for doing so. The first reason is that the matching documents are sorted in descending order by their relevance when using MongoDB's full-text search. For most search use cases, it's hard to imagine a scenario where a user would want to search for and retrieve some items from a database only to see the least relevant items first.

The second reason has to do with how we get MongoDB to sort the documents in order of their relevance. We use the `{ $meta: "textScore" }` expression to sort the most relevant documents first (rather than using a `1` or `-1` to indicate the sort direction on a specific field as we have previously). However, in doing so we allow the `$meta` expression to assume full control over the sort order, which is descending only. There is no simple way to flip the sort order of these documents as we have done with other paginated queries.

Ultimately, hacking around this MongoDB default doesn't make much sense because, again, it's very hard to imagine a scenario where a user would want to see the least relevant search results first. Additionally, and for the same reasons, the `searchProfiles` query omits what becomes a redundant `orderBy` argument.

We also can't rely on the basic `find` method to implement paginated search results. We will need to use a more advanced feature of MongoDB called *aggregation*. As defined by the MongoDB documentation:

Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

A key point to take away from this definition is that aggregations are quite powerful and can be used to group documents and perform operations on them to derive data and insights from the database that go beyond what's explicitly contained in the documents themselves. Aggregation works by creating a *pipeline* that documents enter and are subsequently transformed in various stages until a final result is output at the end. If we were to replicate a basic query for members of a user's network as an aggregation, then it might look something like this:

```
await this.Model.aggregate([
  { $match: { _id: { $in: [ "_id_1", "_id_2", ... ] } } },
  { $sort: { username: 1 } },
  { $limit: 20 }
]);
```

We pass the pipeline as an array argument to the `aggregate` method to describe the ordered sequence of data aggregation stages to apply to documents in the database. The `$match` stage is like creating a query document for a regular MongoDB query operation—it will pass only the documents that match the specified filter to the next pipeline stage. The `$match` stage is then followed by a `$sort` and a `$limit` stage before returning the matching documents.

Typically, we wouldn't use aggregation for a basic query like this, but it will be useful here where we deal with the tricky business of paginating a full-text search query in MongoDB. To set up the basic pipeline, we'll build out another method in `Pagination.js` called `_getSearchPipeline`:

shared/src/utils/Pagination.js

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async _getSearchPipeline(fromCursorId, filter, first, operator, sort) {
    const textSearchPipeline = [
      { $match: filter },
      { $addFields: { score: { $meta: "textScore" } } },
      { $sort: sort }
    ];

    if (fromCursorId) {
      // add a `$match` stage to get results after a specific cursor...
    }

    textSearchPipeline.push({ $limit: first });

    return textSearchPipeline;
  }

  // ...
}

export default Pagination;
```

This method returns an array of the pipeline stages that we'll later use to pass into the aggregate method. We first match the documents based on the `filter` with a shape of `{ $text: { $search: "some search string" } }`. Next, we use `$addFields` to indicate that we want to keep all existing fields from the input documents and also add the `score` field to each document. You can think of `$addFields` as an all-inclusive projection that also provides an opportunity to add more fields to the documents at this stage. We then sort the documents by relevance, and we also add a `$limit` stage at the very end of the pipeline, just before it's returned from the method.

If the `fromCursorId` argument is defined, then we will first find the `fromDoc` as we did in `_getFilterWithCursor`, but this time also add the text search to the query document and specify a projection document that includes the `score`.

If there are no errors while retrieving the document, then we add another `$match` stage to the pipeline that will return documents where the `score` is either greater than or less than the `fromDoc` `score`, or where the `score` is equal but the `ID` is greater than or less than the `_id` of the `fromDoc`. When looking ahead to the next page of results, the `operator` will be `$lt` because the

documents are sorted in descending order by score and `_id`. Conversely, when looking back for a previous page of results, the operator will be `$gt`:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
    // ...

    async _getSearchPipeline(fromCursorId, filter, first, operator, sort) {
        // ...

        if (fromCursorId) {
            const fromDoc = await this.Model.findOne({
                ...filter,
                _id: fromCursorId
            })
                .select({ score: { $meta: "textScore" } })
                .exec();

            if (!fromDoc) {
                throw new UserInputError(`No record found for ID
                    '${fromCursorId}'`);
            }

            textSearchPipeline.push({
                $match: {
                    $or: [
                        { score: { [operator]: fromDoc._doc.score } },
                        {
                            score: { $eq: fromDoc._doc.score },
                            _id: { [operator]: fromCursorId }
                        }
                    ]
                }
            });
        }

        // ...
    }

    // ...
}
```

```
}

export default Pagination;
```

Before we can update the `ProfilesDataSource` we need to do some refactoring of the `Pagination` class to check if a particular query is a search operation. If it is, then we'll use the aggregation we just set up instead of a regular `find` query. To do that, we'll finish writing the `_isSearchQuery` method, which is the last method we need to create in this class. This method is a simple helper that looks at the sort expression to determine if the documents are sorted on `score` (because documents are only sorted on `score` for search queries):

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  _isSearchQuery(sort) {
    const fieldArr = Object.keys(sort);
    return fieldArr.length && fieldArr[0] === "score";
  }

  // ...
}

export default Pagination;
```

Now we can update the `_getOperator` method to check if the sorting uses a `{ score: { $meta: "textScore" } }` expression instead of integer sort value. We will use the `options` parameter to pass in an optional flag to let the method know that we're dealing with a search query sorted by `textScore`.

If we are handling a search query, then we want to check if any documents exist with a `score` and `_id` greater than those of the `startCursor` on the current page of results for a `_getHasPreviousPage` check. Otherwise, we'll look for documents with `score` and `_id` less than the `endCursor` of the current page:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
```

```
// ...

_getOperator(sort, options = {}) {
  const orderArr = Object.values(sort);
  const checkPreviousTextScore = options?.checkPreviousTextScore
    ? options.checkPreviousTextScore
    : false;
  let operator;

  if (this._isSearchQuery(sort)) {
    operator = checkPreviousTextScore ? "$gt" : "$lt";
  } else {
    operator = orderArr.length && orderArr[0] === -1 ? "$lt" : "$gt";
  }

  return operator;
}

// ...
}

export default Pagination;
```

Now we need to update the `getEdges`, `_getHasNextPage`, and `_getHasPreviousPage` methods to use `_getOperator`. In `getEdges`, we will first add a check to see if we're dealing with a search query using `_isSearchQuery`, and then add an additional error check to prevent backward pagination on a search query:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getEdges(queryArgs) {
    const { after, before, first, last, filter = {}, sort = {} } =
      queryArgs;
    const isSearch = this._isSearchQuery(sort);
    let edges;

    if (!first && !last) {
```

```
    throw new UserInputError(
      "Provide a `first` or `last` value to paginate this connection."
    );
} else if (first && last) {
  throw new UserInputError(
    "Passing `first` and `last` arguments is not supported with this
    connection."
  );
} else if (first < 0 || last < 0) {
  throw new UserInputError(
    "Minimum record request for `first` and `last` arguments is 0."
  );
} else if (first > 100 || last > 100) {
  throw new UserInputError(
    "Maximum record request for `first` and `last` arguments is 100."
  );
} else if (!first && isSearch) {
  throw new UserInputError("Search queries may only be paginated
    forward.");
}

// ...
}

// ...
}

export default Pagination;
```

We can now update the `if / else` block to include a check for a `true` value of `isSearch`. We'll add this at the very top by changing `if (first)` to `else if (first)`, get the operator as usual, get the aggregation pipeline, call the `aggregate` method on the model, and then map over the returned documents to set edges to the correct shape:

`shared/src/utils/Pagination.js`

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async getEdges(queryArgs) {
```

```
// ...

if (isSearch) {
  const operator = this._getOperator(sort);
  const pipeline = await this._getSearchPipeline(
    after,
    filter,
    first,
    operator,
    sort
  );

  const docs = await this.Model.aggregate(pipeline);

  edges = docs.length
    ? docs.map(doc => ({ node: doc, cursor: doc._id }))
    : [];
} else if (first) {
  // ...
} else {
  // ...
}

return edges;
}

// ...
}

export default Pagination;
```

If `after` is `undefined`, then we won't have any issues because the `_getSearchPipeline` method checks for this before attempting to find a `fromDoc`. Next, we can update the `_getHasNextPage` method to use the aggregation for search queries:

*shared/src/utils/Pagination.js*

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...
```

```
async _getHasNextPage(endCursor, filter, sort) {
  const isSearch = this._isSearchQuery(sort);
  const operator = this._getOperator(sort);
  let nextPage;

  if (isSearch) {
    const pipeline = await this._getSearchPipeline(
      endCursor,
      filter,
      1,
      operator,
      sort
    );

    const result = await this.Model.aggregate(pipeline);
    nextPage = result.length;
  } else {
    const queryDoc = await this._getFilterWithCursor(
      endCursor,
      filter,
      operator,
      sort
    );

    nextPage = await this.Model.findOne(queryDoc)
      .select("_id")
      .sort(sort);
  }

  return Boolean(nextPage);
}

// ...
}

export default Pagination;
```

And similarly update the `_getHasPreviousPage` method:

shared/src/utils/Pagination.js

```
import { UserInputError } from "apollo-server";

class Pagination {
  // ...

  async _getHasPreviousPage(startCursor, filter, sort) {
    const isSearch = this._isSearchQuery(sort);
    let prevPage;

    if (isSearch) {
      const operator = this._getOperator(sort, {
        checkPreviousTextScore: true
      });

      const pipeline = await this._getSearchPipeline(
        startCursor,
        filter,
        1,
        operator,
        sort
      );

      const result = await this.Model.aggregate(pipeline);
      prevPage = result.length;
    } else {
      const reverseSort = this._reverseSortDirection(sort);
      const operator = this._getOperator(reverseSort);
      const queryDoc = await this._getFilterWithCursor(
        startCursor,
        filter,
        operator,
        reverseSort
      );

      prevPage = await this.Model.findOne(queryDoc)
        .select("_id")
        .sort(reverseSort);
    }
  }

  return Boolean(prevPage);
}
```

```
    }

    // ...
}

export default Pagination;
```

Notice the change to the call to `_getOperator` now—we pass in the optional `options` object that indicates whether we are dealing with a search query (setting `checkPreviousTextScore` to `true`) and we check if there are any previous results before the current page of search results (using the `$gt` operator for this instead).

At long last, our `Pagination` class is complete! We can now update the `searchProfiles` method in the `ProfilesDataSource` class to use the updated pagination methods:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
    // ...

    async searchProfiles({ after, first, searchString }) {
        const sort = { score: { $meta: "textScore" }, _id: -1 };
        const filter = { $text: { $search: searchString } };
        const queryArgs = { after, first, filter, sort };
        const edges = await this.pagination.getEdges(queryArgs);
        const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

        return { edges, pageInfo };
    }

    // ...
}

export default ProfilesDataSource;
```

Be sure to mark this method as `async` now and also update the parameter to be an object rather than the `searchString` alone. As a final step, we will update the `searchProfiles` resolver in `resolvers.js`:

profiles/src/graphql/resolvers.js

```
// ...  
  
const resolvers = {  
  // ...  
  
  Query: {  
    // ...  
    searchProfiles(parent, { after, first, query }, { dataSources }) {  
      return dataSources.profilesAPI.searchProfiles({  
        after,  
        first,  
        searchString: query  
      });  
    }  
  },  
  
  // ...  
};  
  
export default resolvers;
```

Now test the query:

*GraphQL Query*

```
query SearchProfiles($query: String!, $first: Int) {  
  searchProfiles(query: $query, first: $first) {  
    edges {  
      node {  
        username  
      }  
    }  
    pageInfo {  
      hasPreviousPage  
      hasNextPage  
      startCursor  
      endCursor  
    }  
  }  
}
```

### Query Variables

```
{  
  "query": "ada hopper",  
  "first": 2  
}
```

### API Response

```
{  
  "data": {  
    "searchProfiles": {  
      "edges": [  
        {  
          "node": {  
            "username": "grace_hopper"  
          }  
        },  
        {  
          "node": {  
            "username": "ada_lovelace"  
          }  
        }  
      ],  
      "pageInfo": {  
        "hasPreviousPage": false,  
        "hasNextPage": false,  
        "startCursor": "625ed69a52707218760b0128",  
        "endCursor": "625ed5a652707218760b0122"  
      }  
    }  
  }  
}
```

## Summary

In this chapter, we dove into the mechanics of the Relay Connection Cursor Specification and had the opportunity to implement it against data stored in MongoDB. We began by creating a `Pagination` class containing various generalized methods to pull paginated data from a MongoDB collection.

We then updated the profiles service’s type definitions and used those methods in the `ProfilesDataSource` to paginate the `network` field on the `Profile` type, and then added pagination to the `profiles` and `searchProfiles` fields on the root `Query` type. Paginating the search query also provided a chance to explore an advanced feature of MongoDB called aggregation.

While implementing Relay-style pagination was an lengthy undertaking, all of our hard work here will pay dividends in the next chapter where we’ll reuse the `Pagination` class once again as we build out the bookmarks service.

## Chapter 6

# Bookmark Management with MongoDB and Mongoose

In this chapter, we will:

- Create a new subgraph service to contain type definitions for bookmarks
- Create queries and mutations for reading and writing bookmark data to MongoDB
- Use the `@requires` directive from the Apollo Federation specification to provide additional entity data to an extending subgraph
- Search bookmarks using MongoDB's full-text search

## Install Packages and Configure Mongoose

The bookmarks service—our third subgraph—will finally allow users to save and edit bookmarks. For this service, we'll use MongoDB to persist data again but we'll maintain separation of concerns and create a separate database to contain bookmark documents. Bookmarks will be connected to the `Profile` entity that we defined in Chapter 4 by extending that type to include a `bookmarks` field that will provide a paginated list of a user's saved bookmarks. The `Bookmark` type will also have an `owner` field that will connect it to the profile of the user who created it. Additionally, we'll add a `recommendedBookmarks` field to the `Profile` entity and we'll end up using an advanced feature of Apollo Federation to resolve this computed entity field.

Our first step will be creating a `bookmarks` subdirectory in the root project directory to organize the code for the new service and add a package `.json` file to it:

```
bookmarks/
```

```
npm init --yes
```

Next, we'll install some familiar core packages for the new subgraph:

*bookmarks/*

```
npm i @apollo/subgraph@2.0.3 apollo-datasource@3.3.1 apollo-server@3.7.0  
dotenv@16.0.0 graphql@16.5.0 mongoose@6.3.0
```

As well as nodemon to watch for file changes in the development environment:

*bookmarks/*

```
npm i -D nodemon@2.0.15
```

And again, we will set the type key to module and add a start script:

*bookmarks/package.json*

```
{  
  // ...  
  "type": "module",  
  "scripts": {  
    "dev": "nodemon -r dotenv/config -e env,graphql,js ./src/index.js"  
  },  
  // ...  
}
```

We'll want to make use of the `DateTime` Scalar type, authorization directives, and `Pagination` class in this service as well, so we'll run `npm link graphql` next to reference the `graphql` package installed in the shared directory only:

*bookmarks/*

```
npm link graphql
```

Now we need to add a `.env` file for this service with some familiar environment variables:

*bookmarks/.env*

```
MONGODB_URL=mongodb://127.0.0.1:27017-marked-bookmarks
```

```
NODE_ENV=development  
PORT=4003
```

Note that the `MONGODB_URL` has a slightly different endpoint than the profiles service because we are creating a separate database for the bookmarks data. We also set the `PORT` for the subgraph's GraphQL server endpoint to `4003`.

Once again, we'll need to configure a connection to MongoDB via Mongoose, so we'll create a `src` subdirectory in `bookmarks` and then a `config` subdirectory inside of `src` with a `mongoose.js` file in it with the same code as the one in the profiles service:

`bookmarks/src/config/mongoose.js`

```
import mongoose from "mongoose";

function initMongoose() {
  const connectionUrl = process.env.MONGODB_URL;
  mongoose.connect(connectionUrl);

  mongoose.connection.on("connected", () => {
    console.log(`Mongoose default connection ready at ${connectionUrl}`);
  });

  mongoose.connection.on("error", error => {
    console.log("Mongoose default connection error:", error);
  });
}

export default initMongoose;
```

Now we can create a `Bookmark` model for the collection of bookmark documents. Create a `models` subdirectory in `src` and add a `Bookmark.js` file to that with the following code:

`bookmarks/src/models/Bookmark.js`

```
import mongoose from "mongoose";

const bookmarkSchema = new mongoose.Schema({
  createdAt: {
    type: Date,
    default: Date.now,
    required: true
  },
  ownerAccountId: {
    type: String,
    required: true
  },
  private: {
    type: Boolean,
    default: false,
  }
});

const Bookmark = mongoose.model("Bookmark", bookmarkSchema);

export default Bookmark;
```

```
    required: true
},
tags: [
  {
    type: String
  }
],
title: {
  type: String
},
url: {
  type: String,
  required: true
}
});

bookmarkSchema.index({ title: "text" });

const Bookmark = mongoose.model("Bookmark", bookmarkSchema);

export default Bookmark;
```

The required fields for any bookmark document include the `url` of the bookmarked page, the `ownerAccountId` to indicate what user created it by their Auth0 ID, a `createdAt` time, and `private` field to indicate whether the user wants this bookmark to be visible via their profile. Optionally, a bookmark can also have a `title` and `tags` applied to it.

## Scaffold the Bookmarks Service

Now we're ready to scaffold a few more of the essential files for the bookmarks service. We'll create an `index.js` file in the `src` directory, followed by a `graphql` subdirectory in `bookmarks/src` with `schema.graphql` and `resolvers.js` files, plus a `dataSources` subdirectory in there with a `BookmarksDataSource.js` file in it.

The current file structure in `bookmarks` will look like this:

```
bookmarks
  └── node_modules/
  └── ...
  └── src/
    └── config
      └── mongoose.js
```

```
|   └── graphql
|       └── dataSources
|           └── BookmarksDataSource.js
|       └── resolvers.js
|           └── schema.graphql
|   └── models
|       └── Bookmark.js
|   └── index.js
└── .env
└── package.json
└── package-lock.json
```

Now we can build out the bookmarks service's `index.js` file:

`bookmarks/src/index.js`

```
import { dirname, resolve } from "path";
import { fileURLToPath } from "url";
import { readFileSync } from "fs";

import { ApolloServer, gql } from "apollo-server";
import { buildSubgraphSchema } from "@apollo/subgraph";

import { authDirectives } from "../../shared/src/index.js";
import Bookmark from "./models/Bookmark.js";
import BookmarksDataSource from
  "./graphql/dataSources/BookmarksDataSource.js";
import initMongoose from "./config/mongoose.js";
import resolvers from "./graphql/resolvers.js";

const __dirname = dirname(fileURLToPath(import.meta.url));
const port = process.env.PORT;

const { authDirectivesTypeDefs, authDirectivesTransformer } =
  authDirectives();
const subgraphTypeDefs = readFileSync(
  resolve(__dirname, "./graphql/schema.graphql"),
  "utf-8"
);
const typeDefs = gql(` ${subgraphTypeDefs}\n${authDirectivesTypeDefs}`);
let subgraphSchema = buildSubgraphSchema({ typeDefs, resolvers });
subgraphSchema = authDirectivesTransformer(subgraphSchema);
```

```

const server = new ApolloServer({
  schema: subgraphSchema,
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user };
  },
  dataSources: () => {
    return {
      bookmarksAPI: new BookmarksDataSource({ Bookmark })
    };
  }
});

initMongoose();

server.listen({ port }).then(({ url }) => {
  console.log(`Bookmarks service ready at ${url}`);
});

```

This set-up should look very familiar as it is nearly identical to the one used for the accounts and profiles services. Next, we'll add some boilerplate schema code to `schema.graphql`. We will import both the `@key` and `@shareable` directives here for now:

`bookmarks/src/graphql/schema.graphql`

```

extend schema
@link(url: "https://specs.apollo.dev/federation/v2.0",
      import: ["@key", "@shareable"])

```

And we'll set up the `resolvers.js` file:

`bookmarks/src/graphql/resolvers.js`

```

import { UserInputError } from "apollo-server";

const resolvers = {};

export default resolvers;

```

Next, we'll add some initial code for the `BookmarksDataSource` class. Because a user may have many bookmarks, these lists will be paginated so we'll use the `Pagination` class here again:

`bookmarks/src/graphql/dataSources/BookmarksDataSource.js`

```
import { DataSource } from "apollo-datasource";
import { UserInputError } from "apollo-server";

import { Pagination } from "../../../../../shared/src/index.js";

class BookmarksDataSource extends DataSource {
  constructor({ Bookmark }) {
    super();
    this.Bookmark = Bookmark;
    this.pagination = new Pagination(Bookmark);
  }
}

export default BookmarksDataSource;
```

Now we can jump over to the gateway and add an environment variable for the bookmarks service's endpoint:

`gateway/.env`

```
# ...

ACCOUNTS_ENDPOINT=http://localhost:4001
PROFILES_ENDPOINT=http://localhost:4002
BOOKMARKS_ENDPOINT=http://localhost:4003
```

Next, we'll update the gateway configuration by adding the new subgraph schema to it:

`gateway/src/config/apollo.js`

```
// ...

function initGateway(httpServer) {
  const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: "accounts", url: process.env.ACCOUNTS_ENDPOINT },
        { name: "profiles", url: process.env.PROFILES_ENDPOINT },
        { name: "bookmarks", url: process.env.BOOKMARKS_ENDPOINT }
      ],
      pollIntervalInMs: 1000
    })
  });
}

export default gateway;
```

```
    },
    // ...
});

// ...
```

The new bookmarks service can be started at this point but it won't do much yet. In the next section, we'll add types and resolvers to the new subgraph schema so we can test out the service.

## Add the Bookmark Type and Its Query Fields

The first type we add to the bookmarks service's schema will be the custom `DateTime` Scalar. We'll use it to ensure the `createdAt` time for a bookmark is a valid ISO 8601 date string:

`bookmarks/src/graphql/schema.graphql`

```
# ...

"""
An ISO 8601-encoded UTC date string.
"""

scalar DateTime
```

Thinking one step ahead to when we'll add the `Bookmark` type, there will be an `url` field on that type that we could set to a generic `String` output type, but it would be better if we could bake some validation of this URL string directly into the schema. To do that, we'll add an `URL` custom Scalar type to our shared type definitions. Create a file called `URLType.js` in `shared/src/scalars` and add the following code to it:

`shared/src/scalars/URLType.js`

```
import { ApolloError } from "apollo-server";
import { GraphQLScalarType } from "graphql";
import validator from "validator";

const URLType = new GraphQLScalarType({
  name: "URL",
  description: "A well-formed URL string.",
  parseValue: value => {
    if (validator.isURL(value)) {
      return value;
    }
  }
});
```

```
        throw new ApolloError("String must be a valid URL including a
            protocol");
    },
    serialize: value => {
        if (validator.isURL(value)) {
            return value;
        }
        throw new ApolloError("String must be a valid URL including a
            protocol");
    },
    parseLiteral: ast => {
        if (validator.isURL(ast.value)) {
            return ast.value;
        }
        throw new ApolloError("String must be a valid URL including a
            protocol");
    }
};

export default URLType;
```

As we can see, this logic to handle this custom Scalar type is very similar to `Datetime` because we use the `validator` package again, but here we use it to determine that the supplied string matches an URL-like pattern. We also have to export this new type definition from the module's main `index.js` file:

`shared/src/index.js`

```
import authDirectives from "./directives/authDirectives.js";
import DateTimeType from "./scalars/DateTimeType.js";
import Pagination from "./utils/Pagination.js";
import URLType from "./scalars/URLType.js";

export { authDirectives, DateTimeType, Pagination, URLType };
```

Now we can use the URL Scalar type in the bookmarks service's schema and define the `Bookmark` Object type:

`bookmarks/src/graphql/schema.graphql`

```
# ...
```

```
"""
A well-formatted URL string.
"""

scalar URL

"""

A bookmark contains content authored by a user.
"""

type Bookmark {
    "The unique ID of the bookmark."
    id: ID!
    "The date and time the bookmark was created."
    createdAt: DateTime!
    "Whether a bookmark has been marked as private."
    private: Boolean!
    "User-applied tags for the bookmark."
    tags: [String]
    "A title to describe the bookmarked content."
    title: String
    "The URL of the page to be bookmarked."
    url: URL!
}
```

We haven't connected the `Bookmark` to the user who owns it yet, but we'll address that shortly. For now, let's focus on adding bookmark-related fields on the root `Query` type as well as the types required for paginating lists of bookmarks. Just like profiles, we'll need to add `BookmarkConnection` and `BookmarkEdge` Object types and a `BookmarkOrderBy` Enum type to support Relay-style pagination for bookmark lists. We also need to redefine the  `PageInfo` value type here, and once again, we will need to include the `@shareable` directive on it:

*bookmarks/src/graphql/schema.graphql*

```
# ...

"""

Sorting options for bookmark connections.
"""

enum BookmarkOrderBy {
    "Order bookmarks ascending by creation time."
    CREATED_AT_ASC
    "Order bookmarks descending by creation time."
    CREATED_AT_DESC
```

```
}

# ...

"""
A list of bookmark edges with pagination information.
"""

type BookmarkConnection {
  "A list of bookmark edges."
  edges: [BookmarkEdge]
  "Information to assist with pagination."
  pageInfo: PageInfo!
}

"""

A single bookmark node with its cursor.
"""

type BookmarkEdge {
  "A cursor for use in pagination."
  cursor: ID!
  "A bookmark at the end of an edge."
  node: Bookmark!
}

"""

Information about pagination in a connection.
"""

type PageInfo @shareable {
  "The cursor to continue from when paginating forward."
  endCursor: String
  "Whether there are more items when paginating forward."
  hasNextPage: Boolean!
  "Whether there are more items when paginating backward."
  hasPreviousPage: Boolean!
  "The cursor to continue from them paginating backward."
  startCursor: String
}
```

Now we can define Query fields for retrieving a single bookmark by its ID and searching for relevant bookmarks based on a user-provided search string:

bookmarks/src/graphql/schema.graphql

```
# ...

type Query {
  "Retrieves a single bookmark by ID."
  bookmark(id: ID!): Bookmark!
  "Provides a search string to query relevant bookmarks."
  searchBookmarks(
    after: String
    first: Int
    "The text string to search for in bookmark title."
    query: String!
  ): BookmarkConnection
}
```

Note that none of the bookmark queries will require authentication to view the data, so the data for any bookmark will be publicly queryable unless it has been marked as `private` by the owner. This will allow an unauthenticated user to search for bookmarks in `Marked` as long as they don't try to retrieve information about the bookmark's owner. Other users' data will not be accessible to unauthenticated users when they use a bookmark as an entry point to the API because the reference resolvers for the `Account` and `Profile` types check for a valid access token in the context.

Now we'll connect a bookmark to the user who created it in our schema. To do that, we'll add an `owner` field to the `Bookmark` type with an output type of `Profile`. To use the `Profile` entity in this subgraphs schema we will need to stub out the `Profile` type here as well:

bookmarks/src/graphql/schema.graphql

```
# ...

"""
A bookmark contains content authored by a user.

"""

type Bookmark {
  # ...
  "The profile of the user who authored the bookmark."
  owner: Profile!
  # ...
}

# ...
```

```
"""
A profile contains metadata about a specific user.
"""
type Profile @key(fields: "id") {
  id: ID!
}

# ...
```

But wait! In Chapter 4, we defined the `Profile` entity and set the `fields` argument of the `@key` directive to `id`, just as we did for the `Account` entity. This approach made sense if we think about the entity's key field as acting like a primary key that we can use to identify a unique profile, which would be the MongoDB document ID in this case. However, based on the `Bookmark` model we previously defined for the bookmarks collection in this service's database, a bookmark will only know about its owner's Auth0 field via the `ownerAccountId` field in the document.

Luckily, the profile documents also have the user's unique Auth0 ID saved in a field, so we can use this value to bridge the gap between the `ownerAccountId` field in a bookmark document and the relevant profile document to connect the `Bookmark` and `Profile` types. Apollo Federation can accommodate exactly this kind of scenario with an advanced feature that allows multiple `@key` directives to be applied to an entity type. What's more, it also supports compound keys so we can refer to the `id` field that's nested within the `account` field on the `Profile` type.

We'll need to make a small adjustment in the profiles services to support `Profile` entity resolution via an Auth0 ID. First, we'll update the `schema.graphql` file to include an additional `@key` directive:

`profiles/src/graphql/schema.graphql`

```
# ...

"""
A profile contains metadata about a specific user.
"""
type Profile @key(fields: "id") @key(fields: "account { id }") {
  # ...
}

# ...
```

Additionally, we must update the reference resolver for the `Profile` type in this service to handle representations that include either the profile document's ID or the Auth0 ID:

*profiles/src/graphql/resolvers.js*

```
// ...

const resolvers = {
  // ...

  Profile: {
    __resolveReference(reference, { dataSources, user }) {
      if (user?.sub) {
        return reference.id
          ? dataSources.profilesAPI.getProfileById(reference.id)
          : dataSources.profilesAPI.getProfile({
              accountId: reference.account.id
            });
      }
      throw new ApolloError("Not authorized!");
    },
    // ...
  },
  // ...
};

// ...
};

export default resolvers;
```

Now we can revise the `Profile` definition back in the bookmarks service and use the account-related key instead. And because we use the `Account` type as an output type for the `account` field here, we must also stub out that entity in this subgraph schema:

*bookmarks/src/graphql/schema.graphql*

```
# ...

"""
An account is a unique Auth0 user.
"""

type Account @key(fields: "id") {
  id: ID!
}

# ...
```

```
"""
A profile contains metadata about a specific user.
"""

type Profile @key(fields: "account { id }") {
    account: Account
}

# ...
```

As previously mentioned, we want to be able to traverse the graph in both directions so we can retrieve a paginated list of bookmarks owned by a particular user, so we'll extend the `Profile` type with an additional field next:

`bookmarks/src/graphql/schema.graphql`

```
# ...

"""
A profile contains metadata about a specific user.
"""

type Profile @key(fields: "account { id }") {
    id: ID!
    """

    A list of bookmarks created by the user.

    Private bookmarks will be hidden for non-owners.
    """

    bookmarks(
        after: String
        before: String
        first: Int
        last: Int
        orderBy: BookmarkOrderBy = CREATED_AT_ASC
    ): BookmarkConnection
}

# ...
```

With the bookmark type definitions ready to go we can begin writing methods in the `BookmarksDataSource` to assist with field resolution. First, we'll add a `getBookmarkById` method to get a single bookmark by its document ID. When we retrieve any bookmark we need to be aware of who is requesting it to validate whether they own it and are allowed to view it if it's flagged as private. We check this condition by using MongoDB's `$or` operator:

*bookmarks/src/graphql/dataSources/BookmarksDataSource.js*

```
// ...

class BookmarksDataSource extends DataSource {
    // ...

    getBookmarkById(id, userId) {
        return this.Bookmark.findOne({
            $or: [
                { _id: id, private: false },
                { _id: id, ownerAccountId: userId }
            ]
        }).exec();
    }

    export default BookmarksDataSource;
```

Next, we'll add a `_getBookmarkSort` method to assist with sorting, as well as a `getUserBookmarks` method to fetch bookmarks for a specific user. Again, we selectively show private bookmarks if an authenticated user has triggered the request and they are querying a list of their bookmarks:

*bookmarks/src/graphql/dataSources/BookmarksDataSource.js*

```
// ...

class BookmarksDataSource extends DataSource {
    // ...

    _getBookmarkSort(sortEnum) {
        let sort = {};
        const sortArgs = sortEnum.split("_");
        const direction = sortArgs.pop();
        const field = sortArgs
            .map(arg => arg.toLowerCase())
            .map((arg, i) =>
                i === 0 ? arg : arg.charAt(0).toUpperCase() + arg.slice(1)
            )
            .join("");
        sort[field] = direction === "DESC" ? -1 : 1;
    }
}
```

```
    return sort;
}

// ...

async getUserBookmarks(
  accountId,
  userId,
  { after, before, first, last, orderBy }
) {
  const sort = this._getBookmarkSort(orderBy);
  const filter = {
    $or: [
      { private: false, ownerAccountId: accountId },
      {
        $and: [
          { ownerAccountId: accountId },
          { $expr: { $eq: ["$ownerAccountId", userId] } }
        ]
      }
    ]
  };
  const queryArgs = { after, before, first, last, filter, sort };
  const edges = await this.pagination.getEdges(queryArgs);
  const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

  return { edges, pageInfo };
}

export default BookmarksDataSource;
```

Lastly, we'll add a `searchBookmarks` method to support full-text bookmark search that tailors the results based on an authenticated user:

`bookmarks/src/graphql/dataSources/BookmarksDataSource.js`

```
// ...

class BookmarksDataSource extends DataSource {
  // ...
```

```
async searchBookmarks(userId, { after, first, searchString }) {
  const sort = { score: { $meta: "textScore" }, _id: -1 };
  const filter = {
    $and: [
      {
        $text: { $search: searchString },
        $or: [{ private: false }, { ownerAccountId: userId }]
      }
    ]
  };
  const queryArgs = { after, first, filter, sort };
  const edges = await this.pagination.getEdges(queryArgs);
  const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

  return { edges, pageInfo };
}

export default BookmarksDataSource;
```

The search query above won't account for duplicate URLs that have been saved by multiple users. If you would like to implement an advanced implementation of this query, it could be achieved using another aggregation here instead.

Now that these methods are ready to go, we can finally write our resolvers. We'll start with the Query resolvers:

`bookmarks/src/graphql/resolvers.js`

```
import { UserInputError } from "apollo-server";

const resolvers = {
  Query: {
    async bookmark(root, { id }, { dataSources, user }) {
      const userId = user?.sub ? user.sub : null;
      const bookmark = await dataSources.bookmarksAPI.getBookmarkById(
        id,
        userId
      );
    }
  }
}
```

```
if (!bookmark) {
  throw new UserInputError("Bookmark not available.");
}

return bookmark;
},
searchBookmarks(root, { after, first, query }, { dataSources, user }) {
  const userId = user?.sub ? user.sub : null;

  return dataSources.bookmarksAPI.searchBookmarks(userId, {
    after,
    first,
    searchString: query
  });
}
};

export default resolvers;
```

Next, we'll write the field resolvers for the Bookmark type. Notice that the owner field resolver will need to return a representation of the Profile type in the shape of the nested key field:

*bookmarks/src/graphql/resolvers.js*

```
import { UserInputError } from "apollo-server";

const resolvers = {
  Bookmark: {
    id(bookmark) {
      return bookmark._id;
    },
    owner(bookmark) {
      return { account: { id: bookmark.ownerAccountId } };
    }
  },
  // ...
};

export default resolvers;
```

We also need to write a field resolver for the bookmarks field in the extended Profile type:

bookmarks/src/graphql/resolvers.js

```
import { UserInputError } from "apollo-server";

const resolvers = {
  // ...

  Profile: {
    bookmarks(profile, args, { dataSources, user }) {
      const userId = user?.sub ? user.sub : null;

      return dataSources.bookmarksAPI.getUserBookmarks(
        profile.account.id,
        userId,
        args
      );
    },
  },
  // ...
};

export default resolvers;
```

Lastly, we'll add the import for the `DateTimeType` and `URLType` resolvers at the top of `resolvers.js` so we can use them to resolve the custom Scalar types in this schema:

bookmarks/src/graphql/resolvers.js

```
import { UserInputError } from "apollo-server";

import { DateTimeType, URLType } from "../../../../../shared/src/index.js";

const resolvers = {
  DateTime: DateTimeType,
  URL: URLType,
  // ...
};

export default resolvers;
```

We're almost ready to test our bookmark-related queries now, but before we can do that we need to add some bookmarks to the database. For that, we'll need a `createBookmark` mutation.

## Add a `createBookmark` Mutation

It's time to add some documents to the bookmarks service's database. The `createBookmark` mutation will use an Input Object type for an argument, so we'll create that first:

*bookmarks/src/graphql/schema.graphql*

```
# ...

"""
Provides data to create a bookmark.
"""


```

To create a new bookmark we only require the Auth0 ID of the owner's account and the URL of the page. The rest of the fields will be nullable. The `createBookmark` mutation will look like this:

*bookmarks/src/graphql/schema.graphql*

```
# ...

type Mutation {
  "Creates a new bookmark."
  createBookmark(input: CreateBookmarkInput!): Bookmark!
    @owner(argumentName: "input.ownerAccountId")
}
```

Now we can write a `createBookmark` method in the `BookmarksDataSource` class. As with the tag-like strings that we formatted for the `interests` field in the profile documents, we will standardize the string formats for the list of bookmark tags that a user provides using a similar `_formatTags` helper. When creating a bookmark, we want to ensure that a user only creates a bookmark for a particular URL once, so before we write the new bookmark to the database we first check if we can match another document on the `ownerAccountId` and `url` fields. We'll throw an error if one is found, but otherwise, we'll proceed with creating the new bookmark document:

`bookmarks/src/graphql/dataSources/BookmarksDataSource.js`

```
// ...

class BookmarksDataSource extends DataSource {
    // ...

    _formatTags(tags) {
        return tags.map(tag => tag.replace(/\s+/g, "-").toLowerCase());
    }

    // ...

    async createBookmark(bookmark) {
        const existingBookmarkForUrl = await this.Bookmark.findOne({
            ownerAccountId: bookmark.ownerAccountId,
            url: bookmark.url
        }).exec();

        if (existingBookmarkForUrl) {
            throw new UserInputError("A bookmark for the URL already exists.");
        }

        if (bookmark.tags) {
            const formattedTags = this._formatTags(bookmark.tags);
            bookmark.tags = formattedTags;
        }

        const newBookmark = new this.Bookmark(bookmark);
        return newBookmark.save();
    }

    // ...
}
```

```
export default BookmarksDataSource;
```

We'll use this data source method in a new `createBookmark` resolver:

`bookmarks/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...
  Query: {
    // ...
  },
  Mutation: {
    createBookmark(root, { input }, { dataSources }) {
      return dataSources.bookmarksAPI.createBookmark(input);
    }
  }
};

export default resolvers;
```

Now we're ready to test out the new mutation in Explorer:

*GraphQL Mutation*

```
mutation CreateBookmark($input: CreateBookmarkInput!) {
  createBookmark(input: $input) {
    id
    createdAt
    private
    tags
    title
    url
  }
}
```

*Mutation Variables*

```
{
  "input": {
```

```
        "ownerAccountId": "auth0|625b5a7847a7f7006f3ce7ab",
        "private": false,
        "tags": ["graphql"],
        "title": "GraphQL specification",
        "url": "https://spec.graphql.org/"
    }
}
```

#### API Response

```
{
  "data": {
    "createBookmark": {
      "id": "626367bbb1e459abac5eb08e",
      "createdAt": "2022-04-22T13:24:03.755Z",
      "private": false,
      "tags": [
        "graphql"
      ],
      "title": "GraphQL specification",
      "url": "https://spec.graphql.org/"
    }
  }
}
```

With one bookmark in the database, we can test out the bookmark query now. This will be our first operation that spans all three subgraphs:

#### GraphQL Query

```
query Bookmark($id: ID!) {
  bookmark(id: $id) {
    title
    url
    owner {
      username
      account {
        email
      }
    }
  }
}
```

*Query Variables*

```
{  
  "id": "626300d3d1cff70555d1f61f"  
}
```

*API Response*

```
{  
  "data": {  
    "bookmark": {  
      "title": "GraphQL Specification",  
      "url": "https://spec.graphql.org/",  
      "owner": {  
        "username": "marksalot",  
        "account": {  
          "email": "marksalot@markedmail.com"  
        }  
      }  
    }  
  }  
}
```

As we can see from the gateway's query plan, there are three hops required to fully resolve all of the fields for this operation. The gateway's first trip is to the bookmarks service. After all of the Bookmark fields are resolved, it sends parallel requests to the profiles and accounts services to resolve the remaining user-related fields:

*Query Plan*

```
QueryPlan {  
  Sequence {  
    Fetch(service: "bookmarks") {  
      bookmark(id: $bookmarkId) {  
        title  
        url  
        owner {  
          __typename  
          account {  
            __typename  
            id  
          }  
        }  
      }  
    }  
  }  
}
```

```
        }
    }
},
Parallel {
  Flatten(path: "bookmark.owner") {
    Fetch(service: "profiles") {
      {
        ... on Profile {
          __typename
          account {
            id
          }
        }
      } =>
      {
        ... on Profile {
          username
        }
      }
    },
  },
  Flatten(path: "bookmark.owner.account") {
    Fetch(service: "accounts") {
      {
        ... on Account {
          __typename
          id
        }
      } =>
      {
        ... on Account {
          email
        }
      }
    },
  },
},
}
}
```

Add a few more bookmarks using the `createBookmark` mutation now for at least two different users. Make sure that some of those bookmarks are set to private so we can verify MongoDB's

filtering logic for an authenticated user. Now try running the `profile` query again with the `bookmarks` field add to the operation document:

#### *GraphQL Query*

```
query Profile($username: String!, $first: Int) {  
  profile(username: $username) {  
    username  
    bookmarks(first: $first) {  
      edges {  
        node {  
          private  
          tags  
          title  
          url  
        }  
      }  
    }  
  }  
}
```

#### *Query Variables*

```
{  
  "username": "marksalot",  
  "first": 10  
}
```

#### *API Response*

```
{  
  "data": {  
    "profile": {  
      "username": "marksalot",  
      "bookmarks": {  
        "edges": [  
          {  
            "node": {  
              "private": true,  
              "tags": ["graphql", "caching"],  
              "title": "Caching & GraphQL: Setting the Story Straight",  
              "url": "https://youtu.be/CV3puKM_G14/"  
            }  
          }  
        ]  
      }  
    }  
  }  
}
```

```
        },
        {
          "node": {
            "private": false,
            "tags": ["graphql", "apollo"],
            "title": "Apollo Federation docs",
            "url": "https://www.apollographql.com/docs/federation/"
          }
        },
        {
          "node": {
            "private": false,
            "tags": [
              "graphql"
            ],
            "title": "GraphQL Specification",
            "url": "https://spec.graphql.org/"
          }
        }
      ]
    }
  }
}
```

Be sure to verify that you can't see any private bookmarks in the result when you rerun the operation with another user's JWT in the `Authorization` header.

Lastly, we'll test out the `searchQuery`:

#### *GraphQL Query*

```
query SearchBookmarks($query: String!, $first: Int) {
  searchBookmarks(query: $query, first: $first) {
    edges {
      node {
        createdAt
        private
        tags
        title
        url
      }
    }
  }
}
```

```
    }
}
```

### Query Variables

```
{
  "query": "GraphQL",
  "first": 10
}
```

### API Response

```
{
  "data": {
    "searchBookmarks": {
      "edges": [
        {
          "node": {
            "createdAt": "2022-04-22T19:24:03.755Z",
            "private": false,
            "tags": [
              "graphql"
            ],
            "title": "GraphQL Specification",
            "url": "https://spec.graphql.org/"
          }
        },
        {
          "node": {
            "createdAt": "2022-04-22T19:45:54.568Z",
            "private": false,
            "tags": [
              "graphql"
            ],
            "title": "Life of a GraphQL Query – Validation",
            "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query-validation-18a8fb52f1898"
          }
        },
        {
          "node": {
            "createdAt": "2022-04-22T19:45:24.867Z",
            "private": false,
            "tags": [
              "graphql"
            ],
            "title": "How to Write a GraphQL API"
          }
        }
      ]
    }
  }
}
```

```
        "private": false,
        "tags": [
          "graphql"
        ],
        "title": "Life of a GraphQL Query – Lexing/Parsing",
        "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query-l_
               exing-parsing-ca7c5045fad8"
      }
    }
  ]
}
```

For the search query, be sure to confirm that the visibility of private bookmarks changes as expected when swapping the authenticated user's JWT and also removing the Authorization header entirely.

## Add `updateBookmark` and `deleteBookmark` Mutations

Now that we can create and query bookmarks we'll want to give users a way to update and delete them as well. First, we'll address bookmark updates by adding an `UpdateBookmarkInput` Input Object type and an `updateBookmark` field on the root Mutation type:

`bookmarks/src/graphql/schema.graphql`

```
# ...

"""
Provides data to update an existing bookmark.
"""

 UpdateBookmarkInput {
  "The unique ID of the bookmark."
  id: ID!
  "The Auth0 ID of the user who created the bookmark."
  ownerAccountId: ID!
  "The updated privacy setting for the bookmark."
  private: Boolean
  "An updated user-applied tags for the bookmark."
  tags: [String]
  "The updated title to describe the bookmarked content."
}
```

```
    title: String
      "The updated URL of the bookmarked page."
    url: URL
}

# ...

type Mutation {
  # ...
  "Updates a bookmark."
  updateBookmark(input: UpdateBookmarkInput!): Bookmark!
    @owner(argumentName: "input.ownerAccountId")
}
```

We included an `ownerAccountId` field in the `UpdateBookmarkInput` to help facilitate field-level authorization and prevent non-owners from updating bookmarks that they don't own. However, we still have a security vulnerability related to bookmark updates. The `@owner` directive will prevent unauthenticated users from trying to update a bookmark it will also prevent authenticated users from updating bookmarks they don't own if they submit the true `ownerAccountId` value with the mutation. However, there's nothing to prevent an authenticated bad actor from submitting their own `ownerAccountId` instead of the actual bookmark owner's ID. In this instance, the bad actor's `ownerAccountId` would equal the `sub` value of the validated token and the mutation would proceed.

This scenario presents a shortcoming in our directive-based authorization approach that would be more easily addressed using resolver middleware for field-level authorization. Due to the constraints of the `@owner` directive implementation, when we create the `updateBookmark` method in the `BookmarksDataSource` class we will use the bookmark's document ID and the authenticated user's account ID from a validated token to match on when finding the document in MongoDB to update:

`bookmarks/src/graphql/dataSources/BookmarksDataSource.js`

```
// ...

class BookmarksDataSource extends DataSource {
  // ...

  async updateBookmark(
    id,
    userId,
    { ownerAccountId: _, ...updatedBookmarkData } )
  {
```

```
if (
  !updatedBookmarkData ||
  (updatedBookmarkData && Object.keys(updatedBookmarkData).length ===
  0)
) {
  throw new UserInputError("You must supply some bookmark data to
    update.");
}

if (updatedBookmarkData) {
  const formattedTags = this._formatTags(updatedBookmarkData.tags);
  updatedBookmarkData.tags = formattedTags;
}

return this.Bookmark.findOneAndUpdate(
  { _id: id, ownerAccountId: userId },
  updatedBookmarkData,
  {
    new: true
  }
);
}

export default BookmarksDataSource;
```

In the resolvers now, we can add the `updateBookmark` mutation. The `updateBookmark` field on the root `Mutation` type has a non-nullable `Bookmark` as an output type, so if the data source method doesn't return a bookmark because the bookmark ID and the owner's account ID were mismatched, then we'll throw and a `UserInputError`:

*bookmarks/src/graphql/resolvers.js*

```
// ...

const resolvers = {
// ...

Mutation: {
  createBookmark(root, { input }, { dataSources }) {
    return dataSources.bookmarksAPI.createBookmark(input);
  },
}
```

```
async updateBookmark(
  root,
  { input: { id, ...rest } },
  { dataSources, user }
) {
  const userId = user?.sub ? user.sub : null;
  const bookmark = await dataSources.bookmarksAPI.updateBookmark(
    id,
    userId,
    rest
  );

  if (!bookmark) {
    throw new UserInputError("Bookmark cannot be updated.");
  }

  return bookmark;
}
};

export default resolvers;
```

The `updateBookmark` mutation is now ready to test on an existing bookmark:

#### *GraphQL Mutation*

```
mutation UpdateBookmark($input: UpdateBookmarkInput!) {
  updateBookmark(input: $input) {
    id
    tags
    title
    url
  }
}
```

#### *Mutation Variables*

```
{
  "input": {
    "id": "6263010dd1cff70555d1f622",
    "ownerAccountId": "auth0|625b5a7847a7f7006f3ce7ab",
```

```
        "tags": ["graphql", "apollo", "federation-2"]
    }
}
```

*API Response*

```
{
  "data": {
    "updateBookmark": {
      "id": "6263010dd1cff70555d1f622",
      "tags": [
        "graphql",
        "apollo",
        "federation-2"
      ],
      "title": "Apollo Federation docs",
      "url": "https://www.apollographql.com/docs/federation/"
    }
  }
}
```

To wrap up, we'll add a `deleteBookmark` mutation to the subgraph schema now:

`bookmarks/src/graphql/schema.graphql`

```
# ...

"""
Provides data to delete a bookmark.

"""
input DeleteBookmarkInput {
  "The unique ID of the bookmark."
  id: ID!
  "The Auth0 ID of the user who created the bookmark."
  ownerAccountId: ID!
}

# ...

type Mutation {
  # ...
  "Deletes a bookmark."
}
```

```
deleteBookmark(input: DeleteBookmarkInput!): Boolean!
  @owner(argumentName: "input.ownerAccountId")
# ...
}
```

Now we'll add a method to the `BookmarksDataSource` class to handle bookmark deletion. Similar to the `updateBookmark` method, we need to use the authenticated user's account ID as one of the fields to match in the filter document passed to `findOneAndDelete` to prevent users from deleting other user's bookmarks:

`bookmarks/src/graphql/dataSources/BookmarksDataSource.js`

```
// ...

class BookmarksDataSource extends DataSource {
// ...

async deleteBookmark(id, userId) {
  try {
    const deletedBookmark = await this.Bookmark.findOneAndDelete({
      _id: id,
      ownerAccountId: userId
    }).exec();

    return deletedBookmark ? true : false;
  } catch {
    return false;
  }
}

export default BookmarksDataSource;
```

Lastly, we'll add the corresponding resolver:

`bookmarks/src/graphql/resolvers.js`

```
// ...

const resolvers = {
// ...
```

```
Mutation: {
  // ...
  deleteBookmark(root, { input: { id } }, { dataSources, user }) {
    const userId = user?.sub ? user.sub : null;

    return dataSources.bookmarksAPI.deleteBookmark(id, userId);
  },
  // ...
}
};

export default resolvers;
```

Now we can try deleting an authenticated user's existing bookmark:

#### *GraphQL Mutation*

```
mutation DeleteBookmark($input: DeleteBookmarkInput!) {
  deleteBookmark(input: $input)
}
```

#### *Mutation Variables*

```
{
  "input": {
    "id": "62630650d1cff70555d1f634",
    "ownerAccountId": "auth0|625c652808eac7006851b39f"
  }
}
```

#### *API Response*

```
{
  "data": {
    "deleteBookmark": true
  }
}
```

## Get Recommended Bookmarks Based on a User's Interests

The final addition to the bookmarks service's schema in this chapter will be a new field on the extended `Profile` entity that recommends other users' bookmarks to a user that are based on the interests saved in their profile. Here we are faced with the interesting challenge of getting access to the tag-like list of interests saved in the profiles service's database without querying it directly. Once again, the Apollo Federation specification accounts for this type of advanced scenario and allows us to pass non-key field data from one service to another during query plan execution by using the `@requires` directive. With this additional data available, we can resolve the computed field from within the bookmarks service.

To add the `recommendedBookmarks` field, we must first add the `interests` field to the `Profile` type with an `@external` directive applied to it. The `@external` directive indicates that this subgraph knows that the `interests` field is defined on the entity elsewhere and will reference it here, but it's not capable of resolving that field. But before we can use either of these Federation 2 directives, we must add them to the `import` argument on the `@link` directive for this subgraph schema first. After that, we can add the `recommendedBookmarks` field and apply the `@requires` directive to it with a `fields` argument set to `interests`:

`bookmarks/src/graphql/schema.graphql`

```
extend schema
  @link(url: "https://specs.apollo.dev/federation/v2.0",
    import: ["@external", "@key", "@requires", "@shareable"])

# ...

type Profile @key(fields: "account { id }") {
  account: Account
  interests: [String] @external
  #
  """
  A list of recommended bookmarks for a user, based on their interests.
  """
  recommendedBookmarks(after: String, first: Int): BookmarkConnection
    @requires(fields: "interests")
}

# ...
```

Next, we'll add a `getRecommendedBookmarks` method that converts the array of tag-like interest strings into a single string that can be used for full-text search in MongoDB. We also filter out private bookmarks and any bookmarks that are owned by the user:

`bookmarks/src/graphql/dataSources/BookmarksDataSource.js`

```
// ...

class BookmarksDataSource extends DataSource {
    // ...

    async getRecommendedBookmarks(accountId, interests, { after, first }) {
        const sort = { score: { $meta: "textScore" }, _id: -1 };
        const searchString = interests
            .map(interest =>
                interest.includes("-")
                    ? `${interest.split("-").join(" ")}` // ...
                    : interest
            )
            .join(" ");
        const filter = {
            $and: [
                {
                    $text: { $search: searchString },
                    ownerAccountId: { $ne: accountId },
                    private: false
                }
            ]
        };
        const queryArgs = { after, first, filter, sort };
        const edges = await this.pagination.getEdges(queryArgs);
        const pageInfo = await this.pagination.getPageInfo(edges, queryArgs);

        return { edges, pageInfo };
    }

    // ...
}

export default BookmarksDataSource;
```

Lastly, we'll add the resolver for the `recommendedBookmarks` field. Because we indicated that the `Profile` entity's `interests` field value will be required to resolve this computed field, we will have access to those values in the first parameter of the resolver function:

`bookmarks/src/graphql/resolvers.js`

```
// ...

const resolvers = {
  // ...

  Profile: {
    bookmarks(profile, args, { dataSources, user }) {
      // ...
    },
    recommendedBookmarks({ account, interests }, args, { dataSources }) {
      return dataSources.bookmarksAPI.getRecommendedBookmarks(
        account.id,
        interests,
        args
      );
    }
  },
};

// ...
};

export default resolvers;
```

We're ready to test the field now. Try adding the `recommendedBookmarks` field to the operation document of a `profile` query:

*GraphQL Query*

```
query Profile($username: String!, $first: Int) {
  profile(username: $username) {
    username
    interests
    recommendedBookmarks(first: $first) {
      edges {
        node {
          tags
          title
          url
        }
      }
    }
  }
}
```

```
        }
    }
}
```

### Query Variables

```
{
  "username": "marksalot",
  "first": 10
}
```

### API Response

```
{
  "data": {
    "profile": {
      "username": "marksalot",
      "interests": [
        "graphql",
        "javascript"
      ],
      "recommendedBookmarks": {
        "edges": [
          {
            "node": {
              "tags": [
                "javascript"
              ],
              "title": "Eloquent JavaScript",
              "url": "https://eloquentjavascript.net/"
            }
          },
          {
            "node": {
              "tags": [
                "graphql"
              ],
              "title": "Life of a GraphQL Query – Validation",
              "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query-validation-18a8fb52f1898"
            }
          },
          {
            "node": {
              "tags": [
                "javascript"
              ],
              "title": "The Definitive Guide to JavaScript Performance Optimization",
              "url": "https://www.smashingmagazine.com/2018/01/the-definitive-guide-to-javascript-performance-optimization/"
            }
          }
        ]
      }
    }
  }
}
```

```
{  
  "node": [  
    {"  
      "tags": [  
        "graphql",  
        "apollo"  
      ],  
      "title": "Life of a GraphQL Query – Lexing/Parsing",  
      "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query_ -lexing-parsing-ca7c5045fad8"  
    }  
  ]  
}  
}  
}  
}
```

You should see that only public bookmarks that have been saved by other users are resolved for the recommendedBookmarks field. When using the @requires directive, examining the gateway's query plan can help us understand how data resolved by one subgraph can flow through to another:

#### Query Plan

```
QueryPlan {  
  Sequence {  
    Fetch(service: "profiles") {  
      profile(username: $username) {  
        __typename  
        account {  
          id  
        }  
        username  
        interests  
      }  
    },  
    Flatten(path: "profile") {  
      Fetch(service: "bookmarks") {  
        ... on Profile {
```

```
    __typename
  account {
    id
  }
  interests
}
} =>
{
  ... on Profile {
    recommendedBookmarks(first: $first) {
      edges {
        node {
          tags
          title
          url
        }
      }
    }
  },
  },
},
}
```

## Summary

At the conclusion of this chapter, we have now set up three separate subgraphs and composed them into the supergraph schema. By building out the bookmarks service, we had the opportunity to reinforce everything that we've learned about subgraph design so far while also exploring some advanced features of Apollo Federation such as using multiple entity keys, compound entity keys, and the `@requires` directive to resolve computed fields.

The remainder of this book will focus on adding some polish to our federated graph architecture to get it closer to a production-ready state. In the next chapter, we'll focus on some key areas of concern when optimizing graph performance and protecting it from bad actors who may try to exploit it.

## Chapter 7

# API Performance and Security Considerations

In this chapter, we will:

- Configure Automatic Persisted Queries with Apollo to optimize network requests
- Protect the GraphQL API from excessively nested queries by limiting query depth
- Use the DataLoader package to batch requests for data from MongoDB and Auth0
- Restrict API discoverability by masking default error messages

## Potential Performance and Security Issues

The Marked GraphQL API has come a long way over the last six chapters, but before we add the fourth and final subgraph we should pause and explore what can be done to make our current implementation more secure and performant.

Up to this point, we haven't taken any intentional steps to optimize the performance of this GraphQL API. While Apollo Gateway's query planning algorithm will do its best to make the fewest network hops possible to different subgraph services while resolving all of the fields requested in an operation, it's up to us to optimize anything else on either side of the gateway that may add avoidable latency to requests.

On the security side, we've made relatively more progress by adding access control via Auth0-supported user authentication and incorporating the `@private` and `@owner` custom directives to handle field-level authorization. However, there are still some key points of vulnerability that we should address before this API is shipped to production. And by taking certain security measures to guard against bad actors, we will simultaneously protect it from non-malicious clients who may inadvertently send requests that place excessive load on the API's backing data sources.

Luckily, we can address many key areas of present concern with small configuration changes to Apollo Gateway or Apollo Server. Specifically, we will focus on:

- Complex queries that result in large POST bodies sent over the wire from client applications
- The uncapped depth to which clients may presently nest fields in an operation
- Duplicate queries sent to Auth0 and MongoDB to resolve the same field data that could instead be batched into fewer requests to the data source
- Exposure of the GraphQL API schema to the outside world

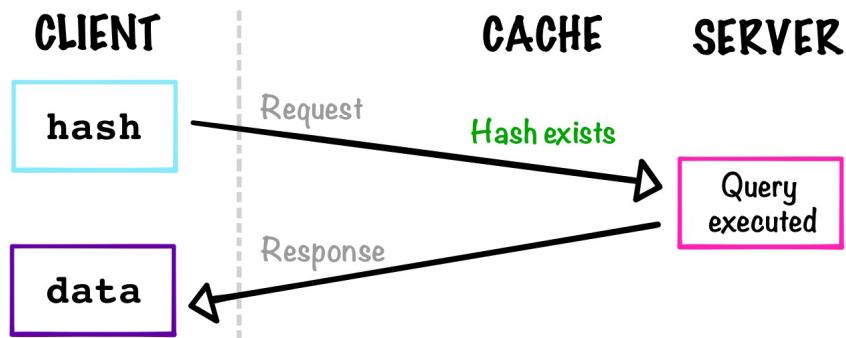
As we address the concerns listed above, do keep in mind that the full list of topics related to optimizing and securing a GraphQL API in production environments could easily fill a book on its own. Our approach will be driven by quick wins that are available in the Apollo software that we're already using (plus a couple of additional packages) as well as some essential "don't leave home without it" best practices for readying GraphQL APIs for production environments. Additional resources for further study are recommended throughout the chapter.

## Configure Automatic Persisted Queries

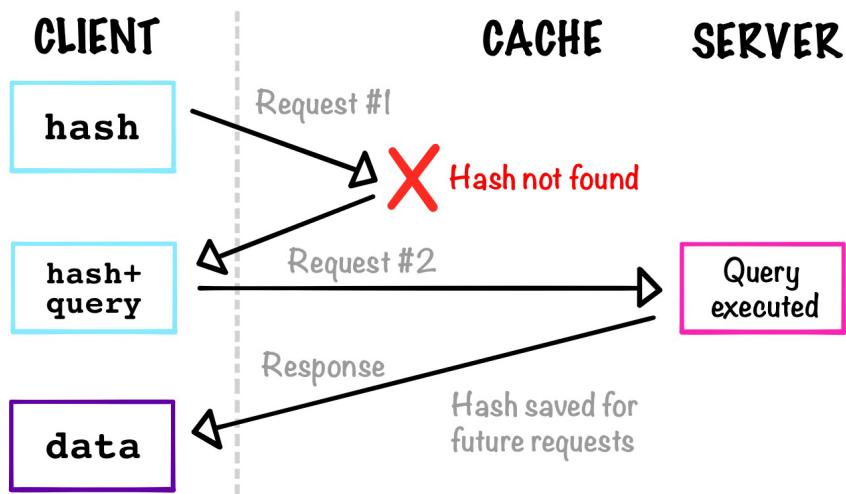
When clients send requests to GraphQL APIs, they often do so using the POST HTTP request method as a default. However, the GraphQL specification doesn't mandate the usage of POST requests. In fact, it doesn't even require that HTTP be used as a transport layer or that serialized JSON be used to represent the operation or format the response.

That said, the POST verb is a common default for GraphQL requests because GET requests have size limits that a complex operation document may exceed. However, relying on POST for this specific reason means that a large amount of data may go over the wire within the context of a single request just to deliver the operation document to the Apollo Server. Additionally, allowing clients to send GET requests would also facilitate the use of browser caches and content delivery networks to optimize API performance further later on.

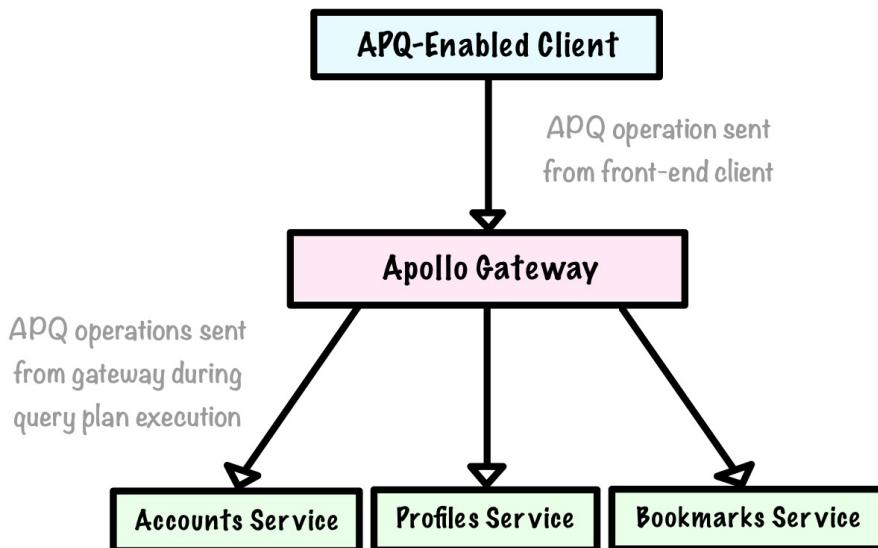
To reduce operation document size and reap the caching-related benefits of GET requests, Apollo offers a solution to optimize the size of requests made to a GraphQL API using what it calls [Automatic Persisted Queries](#), or APQ for short. The premise behind APQ is that rather than always sending the serialized operation document for a specific operation's request payload, we can send a hashed representation of it instead. If the server has seen the hashed operation before, then it will be stored in a cache on the server with the hash as a key and the full document as its value. We can visualize the process as follows:



If the server hasn't seen the hashed version of the operation before, then it will make a subsequent request to the client to get the full document to execute the operation for that request while also caching it for later use:



One of the best parts about APQ is that it's supported in Apollo Server out of the box, so GraphQL clients such as [Apollo Client](#) can be configured to send these hashed queries to help save bandwidth and time for the user when making requests to the gateway. Additionally, because Apollo Gateway also acts as a client to the subgraph's GraphQL servers, we can configure APQ for those requests as well.



We can test if APQ is working using `curl` at the command line. With the gateway up and running, we'll attempt to run a basic `{__typename}` operation via a GET request using its hash with a persisted query:

```
curl -g 'http://localhost:4000/?extensions={"persistedQuery":{"version":1,"sha256Hash":"ecf4edb46db40b5132295c0291d62fb65d6759a9eedfa4d5d612dd5ec54a6b38"}}'
```

We will see the following error message as a response:

```
{"errors": [{"message": "PersistedQueryNotFound", "extensions": {...}}]}
```

We receive this error message because the server hasn't seen this query before, and therefore hasn't persisted the hash and its full operation document value in the cache. If we send the request again with the full query and its hash, then the server will send back the appropriate data and cache the query for future use:

```
curl -g 'http://localhost:4000/?query=__typename&extensions={"persistedQuery":{"version":1,"sha256Hash":"ecf4edb46db40b5132295c0291d62fb65d6759a9eedfa4d5d612dd5ec54a6b38"}}'
```

We will now see the following response:

```
{"data": {"__typename": "Query"}}
```

Now we can try sending the original hashed query again:

```
curl -g 'http://localhost:4000/?extensions={"persistedQuery":{"version":1,"sha256Hash":"ecf4edb46db40b5132295c0291d62fb65d6759a9eedfa4d5d612dd5ec54a6b38"}}'
```

This time, we should see the same data as the query above that included the full operation document instead of the `PersistedQueryNotFound` error response.

As noted, APQ works well with GET requests but it can also help reduce the overall size of requests from clients that continue to use the POST verb too. To illustrate, we'll explore a more complex example. Imagine an operation for a list of profiles that queries some user metadata and related account data for each profile, as well as paginated lists of network members, bookmarks, and recommended bookmarks for each user. The operation document for this query would be lengthy, especially with `PageInfo` fields included for all four paginated list fields. Try running a non-APQ version of the request using this command with a valid JWT included as a header:

```
curl -d '{"operationName": "Profiles", "variables": {}, "query": "query Profiles {\n    profiles(first: 10) {\n        edges {\n            node {\n                username\n                interests\n                account {\n                    email\n                    network(first: 10) {\n                        edges {\n                            node {\n                                username\n                                interests\n                            }\n                        pageInfo {\n                            hasPreviousPage\n                            hasNextPage\n                            startCursor\n                            endCursor\n                        }\n                    }\n                    pageInfo {\n                        hasPreviousPage\n                        hasNextPage\n                        startCursor\n                        endCursor\n                    }\n                    recommendedBookmarks(first: 10) {\n                        edges {\n                            node {\n                                tags\n                                title\n                                url\n                            }\n                            pageInfo {\n                                hasNextPage\n                                startCursor\n                                endCursor\n                            }\n                        }\n                    }\n                    pageInfo {\n                        hasPreviousPage\n                        hasNextPage\n                        startCursor\n                        endCursor\n                    }\n                }\n            }\n        }\n    }\n}' -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIs..." http://localhost:4000 -v
```

In the command output, we can see that the request has a `Content-Length` of 1250 (which is the length of the request body in bytes). If we send the `Profiles` operation again with the APQ data

included in the body of the request, then we can see that the new Content-Length is now 1376 because it also contains the hashed query:

```
curl -d '{"operationName":"Profiles","variables":{},"query":"query
  Profiles {\n    profiles(first: 10) {\n      edges {\n        node {\n          username\n          interests\n          account {\n            email\n            network(first: 10) {\n              edges {\n                node {\n                  username\n                  interests\n                  }\n                pageInfo {\n                  hasPreviousPage\n                  startCursor\n                  endCursor\n                }\n              } {\n                bookmarks(first: 10) {\n                  edges {\n                    node {\n                      tags\n                      title\n                      url\n                    }\n                  } {\n                    pageInfo {\n                      hasNextPage\n                      hasNextPage\n                      startCursor\n                      endCursor\n                    }\n                  }\n                } {\n                  recommendedBookmarks(first: 10) {\n                    edges {\n                      node {\n                        tags\n                        title\n                        url\n                      }\n                    } {\n                      pageInfo {\n                        hasPreviousPage\n                        hasNextPage\n                        startCursor\n                        endCursor\n                      }\n                    }\n                  }\n                }\n              }\n            }\n          }\n        }\n      }\n    }\n  }\n}\n  , "extensions": {"persistedQuery": {"version": 1, "sha256Hash": "0a2ff72d396bdbd1b8dafb5631e72eea7c39c4bfdd9bc7ac3aa9db34cee39bc2"}}, "variables": {}', -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIs..." http://localhost:4000 -v
```

Now that the operation document is cached, we can rerun it by sending the APQ hash alone:

```
curl -d '{"operationName":"Profiles","variables":{},"extensions": {"persistedQuery": {"version": 1, "sha256Hash": "0a2ff72d396bdbd1b8dafb5631e72eea7c39c4bfdd9bc7ac3aa9db34cee39bc2"}}, "variables": {}', -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIs..." http://localhost:4000 -v
```

With a new Content-Length of 169, our Profiles query request body is now 13.5% of its original size. As previously noted, we can also configure the gateway to send APQ requests to the underlying subgraph services during query resolution. Unlike in Apollo Server, this feature is not enabled by default in Apollo Gateway, but we can opt into using it by setting the apq option in the constructor of the `RemoteGraphQLDataSource` in `apollo.js` as follows:

`gateway/src/config/apollo.js`

```
// ...  
  
function initGateway(httpServer) {  
  const gateway = new ApolloGateway({  
    // ...  
    buildService({ url }) {  
      return new RemoteGraphQLDataSource({  
        apq: true,  
        url,  
        // ...  
      });  
    }  
  });  
  
  // ...  
}  
  
export default initGateway;
```

Because all of our subgraphs are implemented using Apollo Server, they will automatically support APQ-style requests from the gateway, so no additional code is required in each subgraph.

Before moving on we should address what we mean by the “cache” when we talk about APQ. By default, this is an in-memory cache that Apollo Server will manage for us behind the scenes. In our current implementation, that means that the gateway and three subgraphs will each manage its in-memory APQ cache within the context of each running process. This option is fine for development purposes but you can easily integrate Redis or Memcached to act as an APQ registry, which would be preferred over the in-memory approach in production environments.

## Limit Query Depth

A key area where performance optimization and security intersect in GraphQL is where we put guardrails in place to limit the amount of data a client can request in a single operation. These requests may be unintentionally malicious, but it is nonetheless important to place some clear constraints on what kinds of operations a client can send on behalf of a user, how much data can be queried at once, and how deeply nested the fields can be in the operation document.

We already put some limitations on what kind of data a client can request by adding authorization directives to the subgraph schemas. We also limited how much data a client can request from a list field by setting a maximum of 100 items per page in the `Pagination` class.

What remains outstanding is that we have no way to limit the amount of data across all fields in an operation. A particular area of concern for us is the depth to which fields are nested. One of the main advantages of using GraphQL is the ability to request data from the API in a shape that makes sense for the client and without making multiple requests for all of the different pieces of the data. However, computationally expensive queries such as the following are possible when this power is left unchecked:

*GraphQL Query*

```
query Profiles {
  profiles(first: 10) {
    edges {
      node {
        username
        network(first: 10) {
          edges {
            node {
              username
              network(first: 10) {
                edges {
                  node {
                    username
                    network(first: 10) {
                      edges {
                        node {
                          username
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The first page of results for this query could return up to 10,000 different usernames at once! To limit the exponential overhead of operations such as this, we can install a readily available package that will allow us to add a validation rule that checks query depth in the gateway before it requests data from the subgraph services:

gateway/

```
npm i graphql-depth-limit@1.1.0
```

We'll import `depthLimit` into the server's `apollo.js` file:

gateway/src/config/apollo.js

```
// ...
import { ApolloServerPluginDrainHttpServer } from "apollo-server-core";
import depthLimit from "graphql-depth-limit";

// ...
```

Next, we'll add a `validationRules` property to the `ApolloServer` options that will set a maximum `depthLimit` of 10:

gateway/src/config/apollo.js

```
// ...

function initGateway(httpServer) {
  // ...

  return new ApolloServer({
    gateway,
    plugins: [ApolloServerPluginDrainHttpServer({ httpServer })],
    context: ({ req }) => {
      const user = req.user || null;
      return { user };
    },
    validationRules: [depthLimit(10)]
  });
}

export default initGateway;
```

If we try running the previous query in Explorer now, we'll get an error message that says '`Profiles`' exceeds maximum operation depth of 10. If we remove the innermost `network` field, then our query will run again without errors.

### What about rate limiting and query cost analysis?

Limiting query depth can be an essential element of a demand control strategy for a GraphQL API, but it may not be enough. For example, you may also need to limit query breadth to set a maximum number of root-level fields that may be included in an operation to prevent various kinds of batching attacks.

However, depth and breadth limiting together may not provide the necessary level of nuance for the overall amount limiting of data in individual API requests. Further, you may need to implement a full rate-limiting strategy for your API if you allow third parties access to it. In these scenarios, you may be better serviced by using *query cost analysis* to assign a total cost to an operation based on the fields requested and then reject certain operations that exceed a maximum set cost per query and also apply that cost against a rate limit.

Where query cost analysis is warranted, common approaches to doing this include adding validation rules to a GraphQL API and using type system directives to annotate parts of the schema with various cost-related metadata. There's no one-size-fits-all approach to doing query cost analysis, but there are some existing open-source solutions that you can use to add support for this to a GraphQL API as long as the assumptions that these packages make on your behalf align with your query cost analysis goals.

IBM has also done conducted some recent research in this area that may be of interest:

- [A Principled Approach to GraphQL Query Cost Analysis \(research paper\)](#)
- [GraphQL Cost Directives \(draft specification document\)](#)

## Batch Database Queries Using DataLoader

Now that we've placed a cap on the depth to which fields can be nested we will optimize how repeated requests for the same data are made within the context of a single query operation. For example, when we search for bookmarks on a particular topic we can also query data about the user who owns each bookmark:

### GraphQL Query

```
query SearchBookmarks($query: String!, $first: Int) {
  searchBookmarks(query: $query, first: $first) {
    edges {
      node {
        title
        url
        owner {
          id
        }
      }
    }
  }
}
```

```
        username
      }
    }
  }
}
```

### Query Variables

```
{
  "query": "graphql",
  "first": 5
}
```

We end up with an *N+1 problem* on our hands here because for each `searchBookmarks` query we end up making “N” additional queries to the database to resolve the `owner` field for those bookmarks. What’s more, a number of these N queries for the owners’ data are potentially requesting the same data too—there’s always a chance that multiple bookmarks from the same user will appear on a single page of search results.

We can see this problem in action by turning on Mongoose’s debug mode:

`profiles/src/config/mongoose.js`

```
import mongoose from "mongoose";

function initMongoose() {
  // ...

  mongoose.set("debug", true);
}

export default initMongoose;
```

When we run the previous `SearchBookmarks` query in Explorer we will see all of the queries made to MongoDB logged to the console. The logs will look something like this:

```
# ...
Mongoose: profiles.findOne({ accountId: 'auth0|625b5a7847a7f7006f3ce7ab'
  }, { projection: {} })
Mongoose: profiles.findOne({ accountId: 'auth0|625b5a7847a7f7006f3ce7ab'
  }, { projection: {} })
```

```
Mongoose: profiles.findOne({ accountId: 'auth0|625c652808eac7006851b39f'
  }, { projection: {} })
Mongoose: profiles.findOne({ accountId: 'auth0|625c652808eac7006851b39f'
  }, { projection: {} })
Mongoose: profiles.findOne({ accountId: 'auth0|625b5a7847a7f7006f3ce7ab'
  }, { projection: {} })
```

In the sample output above, we can see that we're running MongoDB queries to fetch one user three times and another user two times to get the data for the bookmark owners, so this approach to resolving the children of the owner field likely won't scale very well in the future!

Under the hood, Apollo Federation does take care of batching the requests from the bookmarks service to the profiles service per field, so there is only one query made between services to resolve all five profiles for the bookmarks' owner field. However, once that request reaches the profiles service it's up to us to deduplicate and batch queries made to MongoDB for the required profile documents.

The DataLoader library was designed specifically for this purpose and we're going to use it here to optimize this aspect of our data fetching layer. Using DataLoader we can make a few small changes to how profiles are fetched in the `ProfilesDataSource` class and instantly cut the five separate database queries for profiles down to one. First, we'll need to install the DataLoader package in the profiles service:

```
profiles/
```

```
npm i dataloader@2.1.0
```

Next, we'll create new `dataLoaders.js` file in `profiles/src/graphql` to organize the DataLoader-related code, which we'll wrap in an `initDataLoaders` function:

```
profiles/src/graphql/dataLoaders.js
```

```
import DataLoader from "dataloader";

import Profile from "../models/Profile.js";

function initDataLoaders() {
  // DataLoaders will be instantiated here and returned in an object...
}

export default initDataLoaders;
```

To batch our requests, we'll need to instantiate a new `DataLoader` object. The `DataLoader` constructor takes a batch function as an argument, and the batch function will have a keys

parameter that represents the array of MongoDB filter documents corresponding to the profiles we want to retrieve. In some cases, we may need to get documents by their `accountId` field but in others, we may need to use the `_id` field (for example, in the `Profile` type's reference resolver).

Inside the batch function, we will call the `find` method on the `Profile` model object instead of `findOne` or `findById` so we can retrieve all of the profile documents at once. We then map over the original keys array to return a new array containing all of the profile documents in the same order as the filter documents that were passed into the function because maintaining the original order is a requirement of DataLoader. Lastly, we must return the `profileLoader` from the `initDataLoaders` function:

`profiles/src/graphql/dataLoaders.js`

```
// ...

function initDataLoaders() {
  const profileLoader = new DataLoader(async keys => {
    const fieldName = Object.keys(keys[0])[0];
    const fieldValues = keys.map(key => key[fieldName]);
    const profiles = await Profile.find({
      [fieldName]: { $in: fieldValues }
    }).exec();

    return keys.map(key =>
      profiles.find(profile => key[fieldName] ===
        profile[fieldName].toString())
    );
  });

  return { profileLoader };
}

export default initDataLoaders;
```

Note that the code in the batch function above assumes that there will only be one field in the filter document that is used as a key. This approach will satisfy our current use case but a more complicated implementation may be required in the future for filter documents that contain multiple fields. Now we can add the loaders to the resolver context for this subgraph:

`profiles/src/index.js`

```
// ...
import initDataLoaders from './graphql/dataLoaders.js';
// ...
```

```
const server = new ApolloServer({
  schema: subgraphSchema,
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user, loaders: initDataLoaders() };
  },
  // ...
});
```

The reason that we add `loaders` to the context object is that we only want to instantiate the loaders once per request. Now we can make a small adjustment to the `ProfilesDataSource` class to get direct access to the context object there. The parent `DataSource` class exposes an `initialize` method with a `config` object parameter that contains the `context` object inside of it. To make this object available to the `ProfilesDataSource` methods, we'll set it on a `context` property for the class:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
  // ...

  initialize(config) {
    this.context = config.context;
  }

  // ...
}

export default ProfilesDataSource;
```

Now we can update the `getProfile` and `getProfileById` methods so that they use the profile `DataLoader` and call its `load` method while passing the appropriate filter document in as an argument:

`profiles/src/graphql/dataSources/ProfilesDataSource.js`

```
// ...

class ProfilesDataSource extends DataSource {
```

```
// ...

getProfile(filter) {
  return this.context.loaders.profileLoader.load(filter);
}

getProfileById(id) {
  return this.context.loaders.profileLoader.load({ _id: id });
}

// ...
}

export default ProfilesDataSource;
```

If we run the `SearchBookmarks` query again we will see dramatically different output logged to the console, but the same query data will be returned as before in Explorer:

```
# ...
Mongoose: profiles.find({ accountId: { '$in': [
  'auth0|625b5a7847a7f7006f3ce7ab', 'auth0|625c652808eac7006851b39f' ] } },
{ projection: {} })
```

Similarly, if we try running a query that requires the `Profile` entity's reference resolver to be called (such as the `accounts` query with the `profile` field included), then we should also see that the `profileLoader` has optimized how these documents are fetched from MongoDB because either the `getProfile` or `getProfileById` methods from the `ProfilesDataSource` will be called from this resolver too.

We can take our use of DataLoader one step further to also optimize how we request user account data from Auth0. For example, this operation will make N requests to the Auth0 API to fetch account information for each user in the list of profiles:

#### *GraphQL Query*

```
query Profiles($first: Int) {
  profiles(first: $first) {
    edges {
      node {
        username
        account {
          email
        }
      }
    }
  }
}
```

```
        }
    }
}
```

#### *Query Variables*

```
{
  "first": 5
}
```

While there's a 1:1 relationship between profiles and accounts and we know we won't fetch any duplicates, we can still combine the requests for individual accounts into a single call to the Auth0 API. This is an important consideration for performance because it can dramatically cut down on network requests while fetching the required user account data and it will also help prevent our app from needlessly running into Auth0's rate limit due to redundant API requests:

Over in the accounts service, let's install `dataloader` there as well:

*accounts/*

```
npm i dataloader@2.1.0
```

As we did with the profiles service, we'll create a `dataLoaders.js` file in `accounts/src/graphql` with a single `initDataLoaders` function that returns an object containing the instantiated `DataLoader` that will handle account fetching. The batch function will use the Auth0 API's `getUsers` method to fetch the required user accounts by their IDs. Auth0 uses [Lucene query syntax](#) to retrieve users based on search criteria, so our search string must be formed like this:

```
"user_id:auth0|5ddff8b60ac910e2a45574a OR
user_id:auth0|5ddff8b769a2d0ed3af4aff ..."
```

To form this string and retrieve the specified user account data from Auth0, the `initDataLoaders` function will contain the following code:

*accounts/src/graphql/dataLoaders.js*

```
import DataLoader from "dataloader";
import auth0 from "../config/auth0.js";

function initDataLoaders() {
  const accountLoader = new DataLoader(async keys => {
```

```
const q = keys.map(key => `user_id:${key}`).join(" OR ");
const accounts = await auth0.getUsers({ search_engine: "v3", q });

return keys.map(key => accounts.find(account =>
  account.user_id === key)
);
});

return { accountLoader };
}

export default initDataLoaders;
```

The `accountLoader` batch function is more straightforward than the `profileLoader` one was because the `keys` array will only ever contain Auth0 ID strings, rather than the filter document objects that we had to handle to make MongoDB queries. Next, we'll add the new loader to the context in the `index.js` file for this service:

`accounts/src/index.js`

```
// ...
import initDataLoaders from "./graphql/dataLoaders.js";
// ...

const server = new ApolloServer({
  schema: subgraphSchema,
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user, loaders: initDataLoaders() };
  },
  // ...
});
```

And make the `context` available inside the `AccountsDataSource` class:

`accounts/src/graphql/dataSources/AccountsDataSource.js`

```
// ...

class AccountsDataSource extends DataSource {
  // ...

  initialize(config) {
```

```
    this.context = config.context;
}

// ...

}

export default AccountsDataSource;
```

We can now update the `getAccountById` method to use the `DataLoader`:

`accounts/src/graphql/dataSources/AccountsDataSource.js`

```
// ...

class AccountsDataSource extends DataSource {
// ...

    getAccountById(id) {
        return this.context.loaders.accountLoader.load(id);
    }

// ...
}

export default AccountsDataSource;
```

With this code in place, user account data will be fetched with a single query to the Auth0 API when running any operation that requires execution of the `Account` entity's reference resolver.

## Restrict API Discoverability

The final measure that we'll take to protect the Marked GraphQL API in this chapter will be to limit the discoverability of the API. But before we write any code, let's pause and explore what it means for a GraphQL API to be "discoverable."

As a developer, one of the great things about GraphQL is that information about the schema can be queried from the GraphQL API itself via *introspection*. In practice, that means that a GraphQL API exposes a special `__schema` field on the root `Query` type that allows us to request information about the API itself, such as the names of its various types and fields. You can see introspection in action by running the following query in Explorer:

### GraphQL Query

```
{  
  __schema {  
    types {  
      name  
      fields {  
        name  
      }  
    }  
  }  
}
```

Introspection is what makes it possible for tools such as Explorer to know all about our GraphQL schema by simply sharing the API's endpoint with it. However, it's considered a best practice to turn off introspection in production for non-public GraphQL APIs to prevent bad actors from learning about the inner workings of your API, including any security vulnerabilities that may have been unknowingly introduced into it.

The `ApolloServer` constructor can take a boolean `introspection` option to indicate whether introspection queries should be allowed against the GraphQL API. By default, introspection is available in Apollo Server unless there is a `NODE_ENV` environment variable set with a value of `production`. In that case, introspection will be automatically disabled. Additionally, if someone tries to navigate directly to our GraphQL endpoint in a browser to access Explorer as we have throughout this book, then Apollo Server will show a production version of this landing page when the `NODE_ENV` is set to `production` because we usually don't want to expose GraphQL IDEs in these environments either. You can provide a custom landing page here instead of the Explorer access link if you like.

Disabling introspection in production environments is an important step toward limiting API discoverability, but it's only the first step. There are other ways that bad actors can learn just as much about our GraphQL schema as they can with an introspection query. One key area of concern is how errors are returned in API responses. Verbose error details are helpful as we develop the GraphQL API but we should be selective about what kind of error-related information we share with the outside world.

Again, Apollo Server provides some coverage related to errors where the `NODE_ENV` is set to either `production` or `test` by eliminating the `exception.stacktrace` data from the `extensions` key of the response. However, there's still more that should be done to prevent details about underlying data sources or the schema from leaking through error messages.

For example, `graphql.js` (a dependency of Apollo Server) likes to provide us with helpful hints in errors messages such as this when you slightly misspell a field:

*API Response*

```
{  
  "errors": [  
    {  
      "message": "Cannot query field \\"email\\" on type \\"Account\\". Did you  
      mean \\"email\\"?",  
      // ...  
    }  
  ]  
}
```

Using an open-source tool such as [clairvoyance](#) plus a generic word list, we can reconstruct our entire API schema from those graphql.js error messages alone through brute-force guessing and testing of field names and without introspection turned on. To eliminate this exposure, we can set the `formatError` option in Apollo Server to remove these hints in production environments:

gateway/src/config/apollo.js

```
// ...  
import { ApolloError, ApolloServer } from "apollo-server-express";  
// ...  
  
function initGateway(httpServer) {  
  // ...  
  
  return new ApolloServer({  
    // ...  
    validationRules: [depthLimit(10)],  
    formatError: err => {  
      if (  
        err.message.includes("Did you mean") &&  
        process.env.NODE_ENV !== "development"  
      ) {  
        return new ApolloError("Internal server error");  
      }  
  
      return err;  
    }  
  });  
  
export default initGateway;
```

Wherever the `NODE_ENV` is set to `production` now, the only details returned for these errors will be an `Internal server error` message:

*API Response*

```
{  
  "errors": [  
    {  
      "message": "Internal server error",  
      "extensions": {}  
    }  
  ]  
}
```

In addition to masking these `graphql.js` error messages, you may also wish to obscure other details than may be returned in an error response. You can consult the Apollo Server documentation for additional details on [masking error messages](#).

On a final note, we've put some important measures in place throughout this chapter to help protect the Marked GraphQL API from malicious queries that are sent either intentionally or unintentionally, but we have only scratched the surface of what's possible. Prior to this chapter, we also called out some important security best practices related to federated GraphQL APIs, such as only allowing the gateway to query subgraphs directly in production environments. It's also a good idea to encrypt traffic between these services too. Beyond these recommendations, you may wish to review the [OWASP GraphQL Cheat Sheet](#) for additional suggestions on how to guard a GraphQL API against exploitation.

## Summary

Throughout this chapter, we made notable strides in improving the performance and security of the Marked GraphQL API. We started by configuring Automatic Persisted Queries to send shorter hashed versions of operations from client applications to the gateway and from the gateway to subgraph services. We also limited the extent to which fields can be nested to guard against malicious queries.

Turning our attention to how data is retrieved from Auth0 and MongoDB, we used the DataLoader package to batch requests for user account and profile data. Lastly, we explored what measures should be taken in production environments to limit the discoverability of our API schema and the underlying data sources behind the subgraphs.

With these enhancements in place, we're now ready to add the final subgraph to the Marked GraphQL API that will help support multi-subgraph workflows.

## Chapter 8

# Multi-Subgraph Workflows with Temporal

In this chapter, we will:

- Explore patterns for coordinating related data operations across distributed services
- Create a subgraph schema for a workflows service and compose it into the supergraph
- Create a new custom type system directive to support machine-to-machine application authorization
- Configure multi-subgraph workflows using Temporal as an orchestrator

## The Challenge of Cascading User Data Deletion Across Services

As a requirement for the Marked application, when a user decides to delete their account then all of their data will be removed from Auth0, MongoDB, and any other backing data sources that may be added in the future. Because Marked is now powered by three different subgraph services, we are faced with the challenge of how to cascade a user's action when deleting their account to deleting their profile and bookmark data as well.

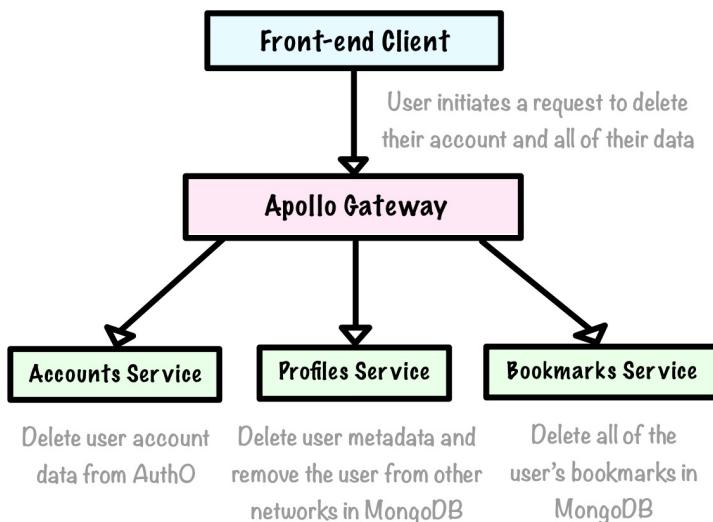
Without this cascade, we could end up with numerous orphaned profile and bookmark documents in their respective databases. Furthermore, from a transparency perspective, a user would typically expect that if they are deleting their account then they can also remove all of their personal information and content at the same time.

At the moment, there are `deleteAccount`, `deleteProfile`, and `deleteBookmark` fields exposed in the API schema that would allow client developers to coordinate this business logic on their side, but this wouldn't be a very GraphQL-like solution to this problem because we typically want to design a schema so that clients can avoid handling precisely this kind of complexity. And in general, it shouldn't be left up to a client application to know what data needs to be cleaned up to prevent orphaned records in a database.

Because we split our server-side application into three self-contained subgraph services at the outset, querying MongoDB directly in the `deleteAccount` method in the `AccountsDataSource` would be an ugly band-aid and violate separation of concerns. The accounts service would then exert control over data that belongs to the profiles and bookmarks services, and do so in a way that is opaque to those services.

What we have finally run into is a common problem within distributed systems where one service needs to know about something that happened in another service, but without introducing hard dependencies between those services. There is nothing built into Apollo Federation that will automatically facilitate a series of mutations apart from the typical serial execution that would happen for top-level `Mutation` fields in any spec-compliant GraphQL server<sup>1</sup>. But again, this would require clients to understand what series of mutations would need to be executed to delete all of the user data and there would be no special error-handling or retry logic automatically available in this approach if something goes wrong during the resolution of one of those fields.

Before we explore potential solutions, let's visualize the work that needs to be done when a user deletes their account data:



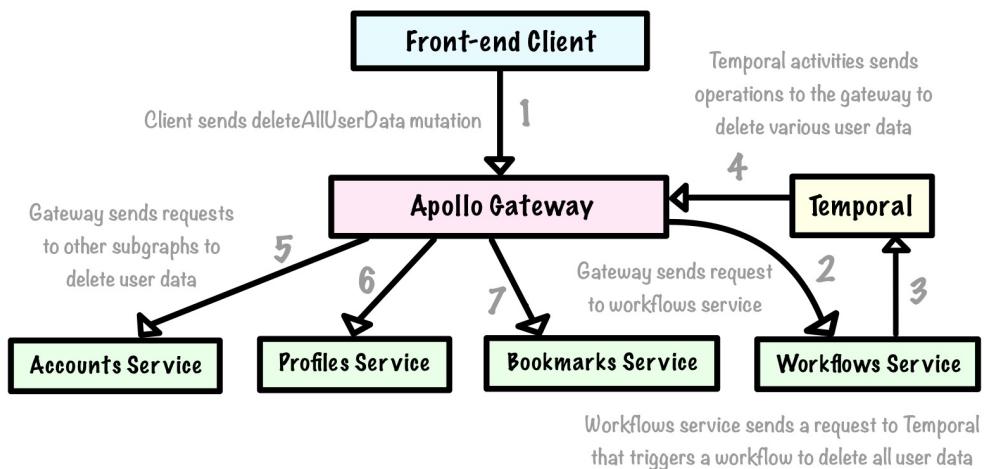
The diagram above may appear simple, but the matter of how to best coordinate interrelated operations on data across distributed systems is not a straightforward one. For Marked specifically, we need to think about what happens if one service fails to execute an operation as expected during the user data deletion process. We won't need to worry about full-blown distributed transactions—where a failure in one subgraph service should trigger a rollback of related operations that were successfully completed in other subgraphs through a *compensating transaction*. However, we should have some retry logic available so that if a subgraph fails to delete some of the user's data for any reason, then the operation can be run again later.

<sup>1</sup>[GraphQL Specification: Normal and Serial Execution](#)

In microservice architectures, *sagas* are a common pattern for coordinating a series of related, local transactions across distributed services<sup>2</sup>. In the first edition of this book, Redis was used as a pub/sub system where subgraph services could subscribe to different queues and run operations based on the messages that they received (for example, deleting a profile in response to an event that indicated that a particular user account had been deleted). This approach was closest to *saga choreography*, where individual services take self-directed actions based on events that are emitted from other services.

In this revised edition, we'll take a different approach and use something closer to *saga orchestration* where a central service will direct the subgraph services to run certain operations in response to some user action. Specifically, we'll use **Temporal** as an orchestrator for the Marked application, and perhaps most interestingly, we will implement Temporal within the context of a new subgraph that will exist solely to coordinate multi-subgraph workflows such as cascading deletion.

Before we start writing any code for this new subgraph, let's paint a high-level picture of how all of the pieces will fit together by the end of this chapter:



As we can see, we will add a `deleteAllUserData` mutation to the API schema via the workflows subgraph and initiate a Temporal workflow from within that field's resolver. The Temporal workflow will contain a series of ordered activities and use a GraphQL client to make calls back to the gateway that will trigger mutations in the three other subgraphs that remove all applicable user data. Once these operations have completed, the workflows subgraph will return a response from the resolver for the `deleteAllUserData` mutation to the gateway and the gateway will then return its response to the front-end client that initiated the request.

Under the hood, all of the required orchestration logic is handled by the workflows subgraph so the client won't have to worry about any back-end implementation details when cleaning up a

<sup>2</sup>Pattern: Saga

user's data. And because Temporal has retry logic built into it, individual subgraphs or clients won't have to worry about implementing retries for specific phases of this workflow either if anything fails.

## Install Packages and Scaffold the Workflows Service

To begin, we'll create a fourth and final subgraph subdirectory called `workflows` in the root project subdirectory and then generate its package.json file:

```
workflows/
```

```
npm init --yes
```

Then we'll install the usual dependencies for the subgraph:

```
workflows/
```

```
npm i @apollo/subgraph@2.0.3 apollo-datasource@3.3.1 apollo-server@3.7.0  
dotenv@16.0.0 graphql@16.5.0 mongoose@6.3.0
```

And add nodemon as a development dependency:

```
workflows/
```

```
npm i -D nodemon@2.0.15
```

We'll set the `type` key to `module` and also add a `dev` script:

```
workflows/package.json
```

```
{  
  // ...  
  "type": "module",  
  "scripts": {  
    "dev": "nodemon -r dotenv/config -e env,graphql,js ./src/index.js"  
  },  
  // ...  
}
```

We'll need to use the `@owner` directive in the workflows service to authorize the `deleteAllUserData` field, so once again, we will link the copy of `graphql` installed in the `shared` directory to avoid errors:

*workflows/*

```
npm link graphql
```

Next, we'll add a `.env` file for this service with a few initial environment variables:

*workflows/.env*

```
GATEWAY_ENDPOINT=http://localhost:4000/
```

```
NODE_ENV=development
```

```
PORT=4004
```

Note that we have set the `GATEWAY_ENDPOINT` variable here so it can be used by a GraphQL client that we will create in this service to make calls back to the gateway in the Temporal activities later on. Next, we'll create some familiar files for this subgraph service. We'll create an `index.js` file in a `src` directory, followed by a `graphql` subdirectory in `workflows/src` with `schema.graphql` and `resolvers.js` files, plus a `dataSources` subdirectory in there with a `WorkflowsDataSource.js` file. The current file structure will now look like this:

```
workflows
  └── node_modules/
      └── ...
  └── src/
      └── graphql
          └── dataSources
              └── WorkflowsDataSource.js
          └── resolvers.js
          └── schema.graphql
      └── index.js
  └── .env
  └── package.json
  └── package-lock.json
```

Now we can build out the workflows service's `index.js` file:

*workflows/src/index.js*

```
import { dirname, resolve } from "path";
import { fileURLToPath } from "url";
import { readFileSync } from "fs";
```

```
import { ApolloServer, gql } from "apollo-server";
import { buildSubgraphSchema } from "@apollo/subgraph";

import { authDirectives } from "../../shared/src/index.js";
import resolvers from "./graphql/resolvers.js";
import WorkflowsDataSource from
  "./graphql/dataSources/WorkflowsDataSource.js";

const __dirname = dirname(fileURLToPath(import.meta.url));
const port = process.env.PORT;

const { authDirectivesTypeDefs, authDirectivesTransformer } =
  authDirectives();
const subgraphTypeDefs = readFileSync(
  resolve(__dirname, "./graphql/schema.graphql"),
  "utf-8"
);
const typeDefs = gql(` ${subgraphTypeDefs}\n${authDirectivesTypeDefs}`);
let subgraphSchema = buildSubgraphSchema({ typeDefs, resolvers });
subgraphSchema = authDirectivesTransformer(subgraphSchema);

const server = new ApolloServer({
  schema: subgraphSchema,
  context: ({ req }) => {
    const user = req.headers.user ? JSON.parse(req.headers.user) : null;
    return { user };
  },
  dataSources: () => {
    return {
      workflowsAPI: new WorkflowsDataSource()
    };
  }
});

server.listen({ port }).then(({ url }) => {
  console.log(`Workflows service ready at ${url}`);
});
```

Next, we'll configure this subgraph schema to use Federation 2 semantics, but we won't need to import any federation directives this time:

```
workflows/src/graphql/schema.graphql
```

```
extend schema @link(url: "https://specs.apollo.dev/federation/v2.0")
```

Then we'll set up the `resolvers.js` file:

```
workflows/src/graphql/resolvers.js
```

```
const resolvers = {};  
  
export default resolvers;
```

As well as the `WorkflowsDataSource` class:

```
workflows/src/graphql/dataSources/WorkflowsDataSource.js
```

```
import { DataSource } from "apollo-datasource";  
  
class WorkflowsDataSource extends DataSource {  
  constructor() {  
    super();  
  }  
}  
  
export default WorkflowsDataSource;
```

Now we can go back over to `gateway` and add a new environment variable for the workflows service's endpoint:

```
gateway/.env
```

```
# ...  
  
ACCOUNTS_ENDPOINT=http://localhost:4001  
PROFILES_ENDPOINT=http://localhost:4002  
BOOKMARKS_ENDPOINT=http://localhost:4003  
WORKFLOWS_ENDPOINT=http://localhost:4004
```

Lastly, the gateway configuration needs to be updated to introspect the new subgraph's schema:

gateway/src/config/apollo.js

```
// ...

function initGateway(httpServer) {
  const gateway = new ApolloGateway({
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: "accounts", url: process.env.ACCOUNTS_ENDPOINT },
        { name: "profiles", url: process.env.PROFILES_ENDPOINT },
        { name: "bookmarks", url: process.env.BOOKMARKS_ENDPOINT },
        { name: "workflows", url: process.env.WORKFLOWS_ENDPOINT }
      ],
      pollIntervalInMs: 1000
    }),
    // ...
  });
}

// ...
```

With the basic plumbing in place for the workflows service, we can open a new terminal tab or window and run `npm run dev` to start it up just like the other services.

## Create Another Auth0 Application to Authenticate the Workflows Service

As discussed earlier in this chapter, when the gateway executes a query plan and forwards the `deleteAllUserData` mutation to the workflows subgraph, that subgraph will need to coordinate a series of requests back to the gateway to run mutations that will remove all of the user's data from Auth0 and MongoDB. As a best practice, we will not send these requests directly to the subgraphs, even though the endpoints are accessible in our development environment (recall the recommendation from Chapter 7 that subgraph endpoints should only be exposed to the gateway for security reasons).

All of the subsequent mutations that must be sent to the accounts, profiles, and bookmarks services will require authorization to ensure that bad actors can't delete other people's data. So now we are faced with an interesting challenge—how will requests sent from the workflows service to the gateway be authorized? When an authenticated end user takes an action to initiate the `deleteAllUserData` mutation their request will be accompanied by their JWT in the `Authorization` header, which the gateway will then decode and validate and pass to the workflows services. But simply sending the previously validated JWT back to the gateway when

the follow-up mutations are triggered by Temporal activities would expose a huge security risk if we were to expect the gateway to trust the validity of that token blindly at that point.

Lucky for us, there's a feature of Auth0 that we have already used that will help address this issue. We will create another machine-to-machine application for the workflows service to authenticate its requests to the gateway, and this will be done independently any token the end user may have sent with their request.

Heading back over to the Auth0 dashboard now, we need to add some special permissions to the Marked GraphQL API that we previously created, then add a new machine-to-machine application for the workflows service and assign those permissions to it, and finally write a rule to filter those permissions out of any JWTs that are sent to human users of the API. First, we'll navigate back to the Marked GraphQL API and add four permissions that will cover all of the use cases for the workflows service when removing user data. Those permissions will include:

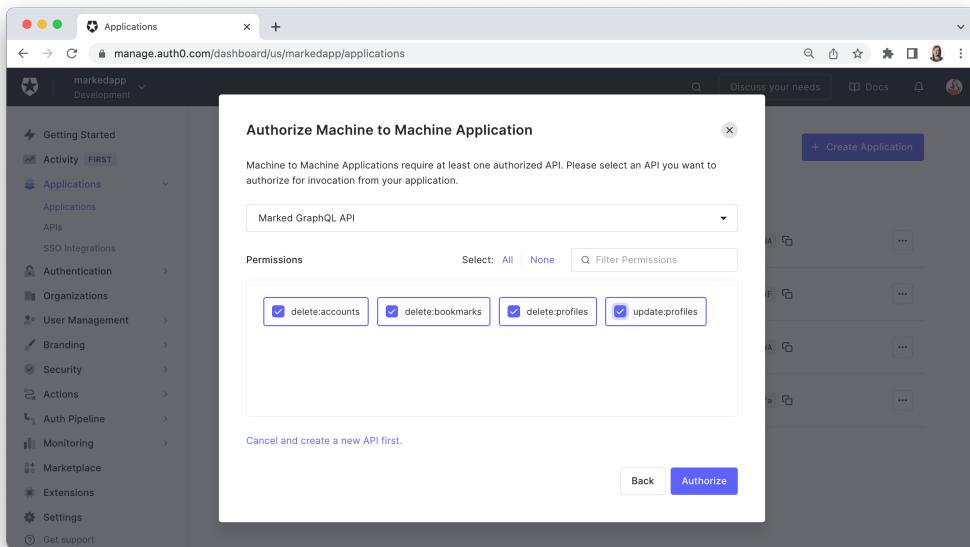
- **delete:accounts:** To authorize deleting a user's account
- **delete:bookmarks:** To authorize deleting all of a user's bookmarks
- **delete:profiles:** To authorize deleting a user's profile
- **update:profiles:** To authorize removing a user from any all other user's networks

Add these permissions to the Auth0 as illustrated:

The screenshot shows the Auth0 API Details interface for the 'Marked GraphQL API'. The left sidebar lists various sections like Getting Started, Activity, Applications, APIs, SSO Integrations, Authentication, Organizations, User Management, Branding, Security, Actions, Auth Pipeline, Monitoring, Marketplace, Extensions, Settings, and Get support. The main content area has a title 'Marked GraphQL API' and sub-sections for Quick Start, Settings, Permissions, Machine to Machine Applications, and Test. The 'Permissions' tab is selected. Under 'Add a Permission (Scope)', there is a field for 'Permission (Scope)' containing 'update:profiles' and a 'Description' field with the value 'Update user profiles'. Below this, under 'List of Permissions (Scopes)', there is a table showing a single entry: 'Permission' is 'delete:accounts' and 'Description' is 'Delete user accounts'.

| Permission      | Description          |
|-----------------|----------------------|
| delete:accounts | Delete user accounts |

Next, we need to create a new machine-to-machine application for the workflows service. Go to the Applications page and create a new application called "Workflows Services" and be sure to select the "Machine to Machine Applications" option. Choose the Marked GraphQL API from the dropdown on the next screen and then assign all of the permissions we just created to it:



Once the application has been created, go to its Settings page and get the Client ID and Client Secret values to add to the workflows service's .env file. We will also add an AUTH0\_AUDIENCE variable here that is set to the gateway endpoint and an AUTH0\_DOMAIN variable that matches the domain of your Auth0 tenant:

`workflows/.env`

```
AUTH0_AUDIENCE=http://localhost:4000/
AUTH0_CLIENT_ID_WORKFLOWS=XXXXXXXXXXXXXXXXXXXX
AUTH0_CLIENT_SECRET_WORKFLOWS=XXXXXXXXXXXXXXXXXXXX
AUTH0_DOMAIN=markedapp.us.auth0.com
```

```
# ...
```

Lastly, we need to address a security issue that we have quietly introduced to the JWTs that Auth0 issues against the Marked GraphQL API. If we leave this issue unchecked, Auth0 will now issue JWTs with payloads that look like this:

```
{
  "iss": "https://markedapp.us.auth0.com/",
  "sub": "auth0|625b5a7847a7f7006f3ce7ab",
  "aud": [
    "http://localhost:4000/",
```

```

    "https://markedapp.us.auth0.com/userinfo"
],
"iat": 1651289203,
"exp": 1651375603,
"azp": "13sVZrbGp8ySe1sHYg0d44D9xxVGMI3F",
"scope": "openid profile email address phone delete:accounts
          delete:bookmarks delete:profiles update:profiles",
"gty": "password"
}

```

By default, any new permissions added to an API will be included in any token's scope field, as we can see above. We only want the workflows service to have access to these custom permissions so we can create a *rule* in Auth0 that provides filters them out from the user JWTs. According to Auth0, “a rule is arbitrary JavaScript code that can be used to extend Auth0’s default behavior when authenticating a user.” You can think of rules as a kind of middleware for Auth0.

To create a rule, access the Rules page via the Auth Pipeline menu item:

The screenshot shows the Auth0 Rules creation interface. On the left, a sidebar navigation includes 'Getting Started', 'Activity FIRST', 'Applications', 'Authentication', 'Organizations', 'User Management', 'Branding', 'Security', 'Actions', 'Auth Pipeline' (which is selected), 'Rules', 'Hooks', 'Monitoring', 'Marketplace', 'Extensions', 'Settings', 'Get support', and 'Give feedback'. The main content area has a title 'Rules' and a sub-instruction: 'A rule is arbitrary JavaScript code that can be used to extend Auth0's default behavior when authenticating a user. Enabled rules will be executed in the order shown below for all users and clients as the final step of the authentication process.' Below this, another instruction says: 'Rules can be used to enrich and transform the user profile, deny access to specific users under certain conditions, retrieve information from external services and much more. To learn more about rules, see <https://auth0.com/docs/customize/rules>'.

The right side of the interface features a large code editor window with the following sample rule script:

```

function (user, context, callback) {
  user.app_metadata = user.app_metadata || {};
  if (context.email === 'mark@markedspace.com') {
    // In this case I check domain
    var index = context.email.indexOf('@');
    if (context.email.substring(index + 1).length > -1) {
      user.app_metadata['domain'] = context.email.substring(index + 1);
    }
  }
  addressToUser(user);
  if (user.app_metadata['domain']) {
    callback(null, user, context);
  } else {
    callback(function(err) {
      if (err) {
        console.log(err);
      }
    });
  }
}

function addressToUser(user) {
  var email = user.email;
  auth0.users.updateAppMetadata(user.user_id, user.app_metadata)
    .then(function(result) {
      console.log(result);
    })
    .catch(function(error) {
      console.error(error);
    });
}

```

In Auth0, rules take the form of an anonymous function with three parameters: `user`, `context`, and `callback`. The `user` parameter is an object that represents the logged-in user as it comes from the identity provider. The `context` object contains additional information about the current authorization transaction (like the user's IP address or whether they tried to authenticate with a mobile device). The `callback` parameter is the callback function that must be called from within the rule to indicate either success or an error. This function sends the potentially modified data

back to Auth0. It's important to remember to call the callback in the rule, otherwise, the script will timeout.

Click the “+ Create” button in the top-righthand corner of the page and choose the “Empty rule” option. We'll add the following code to the Script editor to configure the new rule and then save the changes afterward:

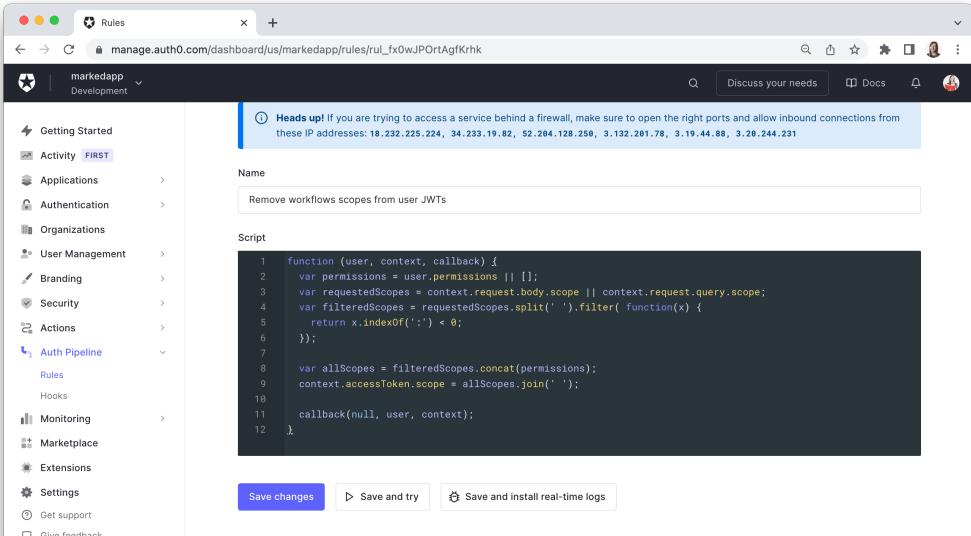
#### Auth0 Script Editor

```
function (user, context, callback) {
  var permissions = user.permissions || [];
  var requestedScopes = context.request.body.scope ||
    context.request.query.scope;
  var filteredScopes = requestedScopes.split(' ').filter( function(x) {
    return x.indexOf(':') < 0;
  });

  var allScopes = filteredScopes.concat(permissions);
  context.accessToken.scope = allScopes.join(' ');

  callback(null, user, context);
}
```

In the Auth0 user interface, the rule code will look like this:



The screenshot shows the Auth0 Rules editor interface. On the left, there's a sidebar with navigation links: Getting Started, Activity (FIRST), Applications, Authentication, Organizations, User Management, Branding, Security, Actions, Auth Pipeline (selected), Rules, Hooks, Monitoring, Marketplace, Extensions, Settings, Get support, and Give feedback. The main area has tabs for 'Rules' (selected) and '+'. The title bar says 'markedapp Development'. The right panel contains a 'Name' field with 'Remove workflows scopes from user JWTs' and a 'Script' field containing the provided JavaScript code. Below the script are three buttons: 'Save changes' (blue), 'Save and try' (gray), and 'Save and install real-time logs' (gray).

Keep in mind that rules are very powerful! You are ultimately responsible for the code you add to this interface and you want to make sure that you don't compromise the security or performance of Auth0 in any way. Review the [rules best practices in the Auth0 documentation](#) before doing further experimentation.

Also, note that there is no version control in the Auth0 script editor, so there's no reverting if you make a change that you're not happy about. There are, however, GitHub and GitLab extensions that will allow you to host your rule code elsewhere.

Now that the custom permissions will no longer be available in the user JWTs, our final task for this section will be configuring a small Auth0 client that can be used from within the workflows service to retrieve a token to send with its requests to the gateway from the Temporal activities that we define later. This client will be defined as an `Auth0Client` class and it will have a single `getToken` method that checks for a non-expired access token in a cache or fetches a new one if required. To facilitate token-fetching and checking an access token's expiration time, we need to install the following two packages in `workflows`:

`workflows/`

```
npm i jwt-decode@3.1.2 node-fetch@3.2.4
```

Now we'll create a `utils` subdirectory in `workflows/src` and add an `Auth0Client.js` file to it with the following code:

`workflows/src/utils/Auth0Client.js`

```
import fetch from "node-fetch";
import jwtDecode from "jwt-decode";

class Auth0Client {
  constructor({ audience, clientId, clientSecret, domain }) {
    this.audience = audience;
    this.clientId = clientId;
    this.clientSecret = clientSecret;
    this.domain = domain;
    this.cache = {};
  }

  getToken = async () => {
    // Token-fetching code will go here...
  };
}
```

```
export default Auth0Client;
```

The token cache will be a basic, in-memory key-value store that caches an access token for the workflows service based on the Auth0 Client ID of the machine-to-machine application we created. To verify that a cached token hasn't expired, we have to decode the token and check its `expiresAt` time. If it's still valid, we return that from the `getToken` method:

workflows/src/utils/Auth0Client.js

```
// ...

class Auth0Client {
    // ...

    getToken = async () => {
        const clientId = this.clientId;
        const cachedToken = this.cache[clientId];

        if (cachedToken) {
            const decodedToken = jwtDecode(cachedToken);
            const expiresAt = decodedToken.exp * 1000;
            const isAuthenticated = expiresAt
                ? new Date().getTime() < expiresAt
                : false;

            if (isAuthenticated) {
                return cachedToken;
            }
        }

        // If the token is expired or doesn't exist get a fresh token...
    };
}

export default Auth0Client;
```

If there isn't a fresh token available in the cache, then we'll fetch a new one from Auth0 using a similar approach to the one in `accounts/src/utils/getToken.js`. This time, however, we'll send the workflows service's application credentials in the body instead of a username and password in `form` data:

workflows/src/utils/Auth0Client.js

```
// ...

class Auth0Client {
    // ...

    getToken = async () => {
        // ...

        const options = {
            method: "POST",
            headers: {
                "cache-control": "no-cache",
                "content-type": "application/json"
            },
            body: JSON.stringify({
                audience: this.audience,
                client_id: this.clientId,
                client_secret: this.clientSecret,
                grant_type: "client_credentials"
            })
        };

        const response = await fetch(`https://${this.domain}/oauth/token`,
            options);
        const body = await response.json();
        const { access_token } = body;

        if (!access_token) {
            throw new Error(
                body.error_description || "Cannot retrieve access token."
            );
        }

        this.cache[this.clientId] = access_token;

        return access_token;
    };
}

export default Auth0Client;
```

Now the workflows service will be able to retrieve an access token for machine-to-machine communication. When successfully fetched, the JWTs will have your machine-to-machine applica-

tion's specific Client ID in the `sub` and `azp` values and it will also have the custom permissions set as the value for the `scope` claim:

```
{  
  "iss": "https://markedapp.us.auth0.com/",  
  "sub": "XXXXXXXXXXXXXXXXXXXX@clients",  
  "aud": "http://localhost:4000/",  
  "iat": 1651330929,  
  "exp": 1651417329,  
  "azp": "XXXXXXXXXXXXXXXXXXXXXX",  
  "scope": "delete:accounts delete:bookmarks delete:profiles  
    update:profiles",  
  "gtv": "client-credentials"  
}
```

We will use these scopes in the next section to create a new custom directive to authorize requests from the workflows service.

## Add a New Authorization Directive to Handle Token Scope

Neither of our existing authorization-related custom directives is designed to handle permissions that are listed in a token's `scope` claim, so we'll need to create a new one and modify the `authDirectivesTransformer` to handle permission validation as well. Over in the `shared` directory, update the `authDirectives.graphql` file to include a new `@scope` directive that accepts a non-nullable list of non-nullable permission strings as an argument:

`shared/src/directives/authDirectives.graphql`

```
# ...  
  
"""  
An authenticated user must have these access token scopes to use the field.  
"""  
  
directive @scope(permissions: [String!]!) on FIELD_DEFINITION
```

We need to verify that a JWT has the correct permissions before executing the field resolver, so we will compare the value of the `scope` field on the validated token in the `context` to the list of applicable permissions that were included as arguments to the directive. The token must have all of the permissions listed in the `permissions` argument of the `@scope` directive for authorization to succeed:

shared/src/directives/authDirectives.js

```
// ...

function authDirectives() {
  const __dirname = dirname(fileURLToPath(import.meta.url));

  return {
    // ...
    authDirectivesTransformer: schema =>
      mapSchema(schema, {
        [MapperKind.OBJECT_FIELD]: fieldConfig => {
          // ...
          const scopeDirective = fieldDirectives.find(
            dir => dir.name === "scope"
          );

          const { resolve = defaultFieldResolver } = fieldConfig;

          if (privateDirective || ownerDirective || scopeDirective) {
            fieldConfig.resolve = function (source, args, context, info) {
              // ...

              let scopeAuthorized = false;
              if (scopeDirective && context.user?.scope) {
                const tokenPermissions = context.user.scope.split(" ");
                scopeAuthorized =
                  scopeDirective.args.permissions.every(scope =>
                    tokenPermissions.includes(scope)
                  );
              }

              if (
                (privateDirective && !privateAuthorized) ||
                (ownerDirective && !ownerArgAuthorized) ||
                (scopeDirective && !scopeAuthorized)
              ) {
                throw new ApolloError("Not authorized!");
              }

              return resolve(source, args, context, info);
            };
          }
        }
      })
  };
}
```

```
        return fieldConfig;
    }
}
);
};

export default authDirectives;
```

Now the subgraph schemas can be updated to use the new directive, and we'll start with the `deleteAccount` mutation in the accounts service:

`accounts/src/graphql/schema.graphql`

```
# ...

type Mutation {
    # ...
    "Deletes an account."
    deleteAccount(id: ID!): Boolean! @scope(permissions: ["delete:accounts"])
    # ...
}
```

Note that we have replaced the `@owner` directive with the `@scope` directive for the `deleteAccount` field now, rather than applying both to the field. We have done this because once we have added the `deleteAllUserData` field to the root `Mutation` type in the workflows service's schema, it will no longer make sense to authorize end users who submit requests from front-end applications to only delete their Auth0 account data. Client developers may not know that they would need to coordinate a series of other mutations to delete and update other applicable data in the bookmarks and profiles services, nor should they have to know this. Moving forward, the `deleteAccount` field will effectively be for internal use only.

Moving on to the profiles services, we will need to update the existing `deleteProfile` field with the new directive and also add a new mutation that removes the deleted user from all other user's networks:

`profiles/src/graphql/schema.graphql`

```
# ...

type Mutation {
    # ...
```

```

    "Deletes a user profile."
deleteProfile(accountId: ID!): Boolean!
  @scope(permissions: ["delete:profiles"])
# ...
"Remove user from other users' networks"
removeUserFromNetworks(accountId: ID!): Boolean!
  @scope(permissions: ["update:profiles"])
# ...
}

```

Similarly, we don't want to allow requests from front-end applications to delete a user profile in isolation, and the `removeUserFromNetworks` mutation is also intended for internal use by the workflows subgraph, so we only apply the `@scope` directive to these fields. Next, we'll add a `removeUserFromNetworks` method to the `ProfilesDataSource` class to support the new field:

`profiles/src/graphql/dataSource/ProfilesDataSource.js`

```

// ...

class ProfilesDataSource extends DataSource {
  // ...

  async removeUserFromNetworks(accountId) {
    try {
      await this.Profile.updateMany(
        { network: { $in: [accountId] } },
        { $pull: { network: accountId } }
      ).exec();
      return true;
    } catch {
      return false;
    }
  }

  // ...
}

export default ProfilesDataSource;

```

And add then the resolver for the new field:

profiles/src/graphql/resolvers.js

```
// ...

const resolvers = {
  // ...

  Mutation: {
    // ...
    removeUserFromNetworks(root, { accountId }, { dataSources }) {
      return dataSources.profilesAPI.removeUserFromNetworks(accountId);
    },
    // ...
  }
};

export default resolvers;
```

Now we'll make the required updates to the bookmarks service. In this case, we want to leave the existing `deleteBookmark` mutation as it is with the `@owner` field applied because Marked users should continue to be allowed to delete individual bookmarks that they created. Instead, we will create a `deleteAllUserBookmarks` mutation that can mass-delete all of the bookmarks that have a particular `ownerAccountId` value associated with them:

bookmarks/src/graphql/schema.graphql

```
# ...

type Mutation {
  # ...
  "Deletes all of a user's bookmarks."
  deleteAllUserBookmarks(ownerAccountId: ID!): Boolean!
    @scope(permissions: ["delete:bookmarks"])
  # ...
}
```

Next, we'll create a new `deleteAllUserBookmarks` method in the `BookmarksDataSource` class to delete all bookmarks with a matching `ownerAccountId` field value:

bookmarks/src/graphql/dataSources/BookmarksDataSource.js

```
// ...
```

```
class BookmarksDataSource extends DataSource {
  // ...

  async deleteAllUserBookmarks(ownerAccountId) {
    try {
      await this.Bookmark.deleteMany({ ownerAccountId }).exec();
      return true;
    } catch {
      return false;
    }
  }

  // ...
}

export default BookmarksDataSource;
```

Lastly, we'll add the resolver:

*bookmarks/src/graphql/resolvers.js*

```
// ...

const resolvers = {
  // ...

  Mutation: {
    // ...
    deleteAllUserBookmarks(root, { accountId }, { dataSources }) {
      return dataSources.bookmarksAPI.deleteAllUserBookmarks(accountId);
    },
    // ...
  }
};

export default resolvers;
```

We have now finished laying the groundwork in the accounts, bookmarks, and profiles services to support cascading deletion of user data, but we won't be able to test the new and updated fields easily quite yet until we configure Temporal in the next section.

## Define a Multi-Subgraph Workflow with Temporal

We have the prerequisite code in place to create the workflow that will delete all of a user's data across the subgraph services when requested. Before we jump in, we'll explore a few key Temporal concepts to contextualize the code that we'll write in this section. If you have used Temporal before, then feel free to skip this overview.

At a high level, Temporal is an orchestration tool for creating long-running workflows that abstract away many of the complications of coordinating this work across distributed services. There are Temporal SDKs for Go, PHP, Java, and TypeScript currently. The TypeScript SDK will work in our JavaScript-based project, but do note that it is still beta at the time of writing so some features aren't yet available in this SDK that are available for other languages (such as compensations to undo partially completed work in a saga).

Temporal has many advanced features, but our work will focus on four key components of its orchestration system:

- **Activities:** Functions that allow Temporal to interact with external resources, such as making a request to the Marked GraphQL API to run a mutation operation.
- **Workflows:** Asynchronous functions that can orchestrate a series of defined activities. These functions should not have side effects.
- **Worker:** A process that connects to the Temporal Server, polls task queues for tasks sent from clients, and runs workflows. We'll need to start up a worker for the workflows service.
- **Clients:** Clients are embedded in application code and connect to the Temporal Server. We'll create a Temporal client inside of the `WorkflowsDataSource` to trigger a new task whenever the `deleteAllUserData` mutation is run.

Before proceeding with the remainder of this section, you may want to watch [Temporal's TypeScript SDK workshop](#) and review its [documentation](#) to gain a broader understanding of the code that will follow.

We will build out each Temporal component for the workflows service in the order of the list above, covering the first three items in this section and then configuring a Temporal client in the `WorkflowsDataSource` in the following section. Starting with activities, we will define an activity function for each mutation that we need to run to delete a user's data. As a reminder, those mutations are `deleteAccount`, `removeUserFromNetworks`, `deleteProfile`, and `deleteAllUserBookmarks`.

We could send these GraphQL operations back to the gateway using a POST request with `fetch`, but we will opt for a higher level of abstraction here and instantiate an `ApolloClient` in the workflows service to handle these requests. If you haven't used Apollo Client for front-end application development before, then not to worry! We will only use a small subset of its core features to handle these requests and you won't need to know any React to do so. Let's install Apollo Client now:

*workflows/*

```
npm i @apollo/client@3.6.1
```

Now we'll create a `client.js` file with the following initial code in it:

*workflows/src/graphq/client.js*

```
import { ApolloClient, InMemoryCache } from "@apollo/client/core/core.cjs";
import { HttpLink } from "@apollo/client/link/http/HttpLink.js";
import fetch from "node-fetch";

function createAuthenticatedApolloClient(uri) {
  if (!uri) {
    throw new Error(
      "Cannot make request to GraphQL API, missing `uri` argument"
    );
  }

  return new ApolloClient({
    cache: new InMemoryCache(),
    link: new HttpLink({ fetch, uri }),
    name: "Workflows Subgraph",
    version: "1.0"
  );
}

export default createAuthenticatedApolloClient;
```

The imports from Apollo Client may look a bit strange because there is [an outstanding issue](#) with the compatibility of its exports with ES modules.

Above, the `ApolloClient` constructor takes a few important options: some metadata about the client name and version, an `inMemoryCache` object, and an `HttpLink` object that we pass our GraphQL endpoint into so Apollo Client knows where to send requests (and how to send them, as indicated by its `fetch` property). We use an `HttpLink` object instead of providing the GraphQL API endpoint directly to Apollo Client because we can chain together multiple link objects here to customize how data is sent to and received from the GraphQL Server.

More specifically, [Apollo Links](#) allow us to modify the way we send requests to a GraphQL API. The most important thing to know about Apollo Links is that they allow us to chain together the units of work we want to perform before getting the result of a GraphQL operation. These units of work

can be composed together so that the first “link” in the chain operates on the original GraphQL operation object and the links that follow work on the output of the link that precedes it.

The last link in the composed chain will be a *terminating link*, and for our purposes, this link will send a network request to the server to fetch a result for the GraphQL operation. This is what the `HttpLink` currently does for us:



Note that a terminating link doesn't necessarily need to fetch data from a server if there is some other way to obtain the desired execution result. A standard Apollo Client set-up will use the `HttpLink` because it creates the typical terminating link for fetching data from a GraphQL endpoint over an HTTP connection.

Apollo Link is key to how we'll fetch an access token from Auth0 for the workflows service so it can be included in the `Authorization` header of the requests made back to the Marked GraphQL API. Because links are composable, we can add another link before the `HttpLink`:

`workflows/src/graphql/client.js`

```

import { ApolloClient, InMemoryCache } from "@apollo/client/core/core.cjs";
import { ApolloLink } from "@apollo/client/link/core/core.cjs";
import { HttpLink } from "@apollo/client/link/http/HttpLink.js";
import { setContext } from "@apollo/client/link/context/context.cjs";
import fetch from "node-fetch";

function createAuthenticatedApolloClient(uri, getToken) {
  // ...

  const authLink = setContext(async (request, { headers }) => {
    let accessToken;

    if (getToken) {
      accessToken = await getToken();
    }

    return {
      headers: {
        ...headers,
        ...(accessToken && { Authorization: `Bearer ${accessToken}` })
      }
    }
  })

  return ApolloClient({
    cache: new InMemoryCache(),
    link: authLink.concat(new HttpLink({ uri }))
  })
}
  
```

```
    };

    });

    return new ApolloClient({
      cache: new InMemoryCache(),
      link: ApolloLink.from([authLink, new HttpLink({ fetch, uri })]),
      name: "Workflows Subgraph",
      version: "1.0"
    });
}

export default createAuthenticatedApolloClient;
```

For good measure, we'll also add an error handling link at the start of the link chain:

*workflows/src/graphql/client.js*

```
// ...
import { onError } from "@apollo/client/link/error/error.cjs";
// ...

function createAuthenticatedApolloClient(uri, getToken) {
  // ...

  const errorLink = onError(({ graphQLErrors, networkError }) => {
    if (graphQLErrors) {
      graphQLErrors.forEach(
        ({ extensions: { serviceName }, message, path }) =>
        console.error(
          `[GraphQL error]: Message: ${message}, Service: ${serviceName},
          Path: ${path[0]}`
        )
    }
    if (networkError) {
      console.error(`[Network error]: ${networkError}`);
    }
  });

  return new ApolloClient({
    cache: new InMemoryCache(),
    link: ApolloLink.from([

```

```
    errorLink,
    authLink,
    new HttpLink({ fetch, uri })
  ],
  name: "Workflows Subgraph",
  version: "1.0"
);
}

export default createAuthenticatedApolloClient;
```

Next, we can write all of the operations that the Temporal activities will send when deleting user data. Create an `operations.js` file in `workflows/src/graphql` and add the following four constants to it:

`workflows/src/graphql/operations.js`

```
import { gql } from "apollo-server";

export const DeleteAccount = gql`mutation DeleteAccount($id: ID!) {
  deleteAccount(id: $id)
}`;

export const DeleteProfile = gql`mutation DeleteProfile($accountId: ID!) {
  deleteProfile(accountId: $accountId)
}`;

export const DeleteAllUserBookmarks = gql`mutation DeleteAllUserBookmarks($ownerAccountId: ID!) {
  deleteAllUserBookmarks(ownerAccountId: $ownerAccountId)
}`;

export const RemoveUserFromNetworks = gql`mutation RemoveUserFromNetworks($accountId: ID!) {
  removeUserFromNetworks(accountId: $accountId)
}`;
```

Note that Apollo Client expects that the operations will be wrapped in the `gql` template tag, as Apollo Server does with the schema SDL. We can now use these operations to create related activities that Temporal will use when executing the user data deletion workflow. Each activity will be a single `async` function that handles a single request to the Marked GraphQL API. If an activity fails, then the workflow will retry it so we must design them to be idempotent and as atomic as possible.

To organize our Temporal-related code, we'll create a `temporal` directory in `workflows/src`. The first file that we add here will be called `activities.js`. The activities will need access to the operations we just defined, the `Auth0Client`, and `createAuthenticatedApolloClient`:

`workflows/src/temporal/activities.js`

```
import {
  DeleteAccount,
  DeleteProfile,
  DeleteAllUserBookmarks,
  RemoveUserFromNetworks
} from "../graphql/operations.js";
import Auth0Client from "../utils/Auth0Client.js";
import createAuthenticatedApolloClient from "../graphql/client.js";

const { getToken } = new Auth0Client({
  audience: process.env.AUTH0_AUDIENCE,
  clientId: process.env.AUTH0_CLIENT_ID_WORKFLOWS,
  clientSecret: process.env.AUTH0_CLIENT_SECRET_WORKFLOWS,
  domain: process.env.AUTH0_DOMAIN
});

const apolloClient = createAuthenticatedApolloClient(
  process.env.GATEWAY_ENDPOINT,
  getToken
);

// Activity functions will go here...
```

Next, we'll define the activity functions. Inside each, we'll call the `mutate` method on the `apolloClient` object and pass it the relevant mutation operation as well as the Auth0 ID of the user whose data should be deleted during the workflow:

*workflows/src/temporal/activities.js*

```
// ...

export async function deleteAccount(id) {
  const response = await apolloClient.mutate({
    mutation: DeleteAccount,
    variables: { id }
  });

  if (response.error) {
    throw new Error(response.error);
  } else {
    return true;
  }
}

export async function deleteProfile(accountId) {
  const response = await apolloClient.mutate({
    mutation: DeleteProfile,
    variables: { accountId }
  });

  if (response.error) {
    throw new Error(response.error);
  } else {
    return true;
  }
}

export async function removeUserFromNetworks(accountId) {
  const response = await apolloClient.mutate({
    mutation: RemoveUserFromNetworks,
    variables: { accountId }
  });

  if (response.error) {
    throw new Error(response.error);
  } else {
    return true;
  }
}
```

```
export async function deleteAllUserBookmarks(ownerAccountId) {
  const response = await apolloClient.mutate({
    mutation: DeleteAllUserBookmarks,
    variables: { ownerAccountId }
  });

  if (response.error) {
    throw new Error(response.error);
  } else {
    return true;
  }
}
```

The activities are ready to go so we can set up the workflow that will execute them when a client adds a task to the queue. To configure a workflow, we'll need to install the Temporal TypeScript SDK in the workflows service now:

*workflows/*

```
npm i @temporalio/temporalio@0.21.x
```

We'll create a `workflows.js` file in `workflows/src/temporal` and use the `proxyActivities` function from the Temporal SDK to create functions that will schedule the deletion-related activities within a `DeleteAllUserData` workflow, which will pass its `accountId` parameter value through to the activity functions:

*workflows/src/temporal/workflows.js*

```
import { proxyActivities } from "@temporalio/workflow";

const {
  deleteAccount,
  deleteProfile,
  deleteAllUserBookmarks,
  removeUserFromNetworks
} = proxyActivities({
  startToCloseTimeout: "1 minute"
});

export async function DeleteAllUserData(accountId) {
  await deleteAccount(accountId);
```

```
    await deleteProfile(accountId);
    await deleteAllUserBookmarks(accountId);
    await removeUserFromNetworks(accountId);
}
```

Next, we'll create a `worker.js` file in `workflows/src/temporal` where we will configure the Temporal worker process that will register all of our workflows and activities and connect to the Temporal Server:

`workflows/src/temporal/worker.js`

```
import { Worker } from "@temporalio/worker";
import { URL } from "url";
import * as activities from "./activities.js";

async function initTemporalWorker() {
  const worker = await Worker.create({
    workflowsPath: new URL("./workflows.js", import.meta.url).pathname,
    activities,
    taskQueue: "marked-app"
  });

  await worker.run();
}

export default initTemporalWorker;
```

The `await worker.run();` line in the `initTemporalWorker` function readies the worker to start accepting tasks on the `marked-app` queue. Lastly, we'll import the `worker` function into the main `index.js` file for this service and start the worker with the subgraph's GraphQL API:

`workflows/src/index.js`

```
// ...
import initTemporalWorker from "./temporal/worker.js";
// ...

initTemporalWorker().catch(error => {
  console.error(error);
  process.exit(1);
});
```

```
server.listen({ port }).then(({ url }) => {
  console.log(`Workflows service ready at ${url}`);
});
```

## Add a `deleteAllUserData` Mutation Field

We're nearly there! All of the code we have written this chapter has been directed at the goal of adding a single `deleteAllUserData` field to the root `Mutation` type in the workflows service's schema. Let's add that field now:

`workflows/src/graphql/schema.graphql`

```
extend schema @link(url: "https://specs.apollo.dev/federation/v2.0")

type Mutation {
  "Deletes all user account, profile, and bookmark data."
  deleteAllUserData(accountId: ID!): Boolean! @owner(argumentName:
    "accountId")
}
```

Over in the `WorkflowsDataSource`, we'll build out the final component that's required to trigger Temporal workflows based on a GraphQL operation. We'll need to import the `Connection` and `WorkflowClient` constructors from the Temporal SDK first, and then we'll use them to create a client so we can schedule new executions in the Temporal Server:

`workflows/src/graphql/dataSources/WorkflowsDataSource.js`

```
import { Connection, WorkflowClient } from "@temporalio/client";
import { DataSource } from "apollo-datasource";

class WorkflowsDataSource extends DataSource {
  constructor() {
    super();
    const connection = new Connection();
    this.client = new WorkflowClient(connection.service);
  }
}

export default WorkflowsDataSource;
```

In the `deleteAllUserData` method, we only need to call the `start` method on the client and pass it some information about the workflow we want to run, the name of the task queue, a unique

ID for the workflow, and an array containing any arguments required by the workflow (the Auth0 account ID in this case):

*workflows/src/graphql/dataSources/WorkflowsDataSource.js*

```
// ...

import { DeleteAllUserData } from "../../temporal/workflows.js";

class WorkflowsDataSource extends DataSource {
// ...

    async deleteAllUserData(accountId) {
        try {
            const handle = await this.client.start(DeleteAllUserData, {
                taskQueue: "marked-app",
                workflowId: `delete-user-data-${accountId}`,
                args: [accountId]
            });
            await handle.result();

            return true;
        } catch {
            return false;
        }
    }
}

export default WorkflowsDataSource;
```

Now we'll add the resolver for the deleteAllUserData field:

*workflows/src/graphql/resolvers.js*

```
const resolvers = {
    Mutation: {
        deleteAllUserData(root, { accountId }, { dataSources }) {
            return dataSources.workflowsAPI.deleteAllUserData(accountId);
        }
    }
};

export default resolvers;
```

We're just about ready to test out the new mutation, but we still need a Temporal Server to process these workflows. You will need to have Docker Engine, the Docker CLI, and Docker Compose installed to run the server. You can refer to the Preface for more information about installing Docker software on your system.

Once Docker is successfully installed and running, you can clone the Temporal Server repository to a preferred location on your computer:

```
git clone https://github.com/temporalio/docker-compose.git temporal-server
```

If you have been using Git for version control up to this point, then you may want to clone the Temporal Server repository outside of the directory that contains the `.git` directory for your project code (unless you feel comfortable working with [Git submodules](#)).

From the terminal, `cd` into the new `temporal-server` directory and run the following command to start up the server:

```
docker compose up
```

It will take a few minutes for the server to start, but once it does, we can also start up the workflows service by opening another terminal window or tab and running `npm run dev` from the `workflows` directory. We can try running the following mutation to delete an existing user and all of their data once the gateway has been restarted and it has composed the new subgraph schema:

#### *GraphQL Mutation*

```
mutation DeleteAllUserData($accountId: ID!) {
  deleteAllUserData(accountId: $accountId)
}
```

#### *Mutation Variables*

```
{
  "accountId": "auth0|626de950dc765f00706b38f4"
}
```

#### *API Response*

```
{
  "data": {
```

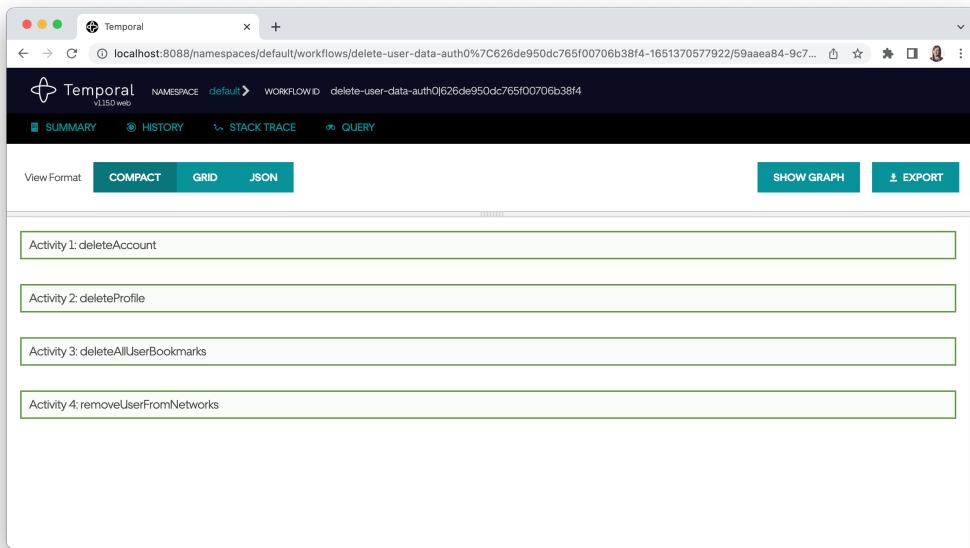
```
        "deleteAllUserData": true  
    }  
}
```

Temporal Server comes with a web user interface built into it that we can use to verify that workflows are completed and view details about activity execution. If we visit <http://localhost:8088> while Temporal Server is running, we can see a list of our in-progress, completed, and failed workflows:

The screenshot shows the Temporal Server web interface at <http://localhost:8088/namespaces/default/workflows?range=last-10-minutes&status=ALL>. The interface has a dark theme with a header bar containing the Temporal logo, the namespace 'default', and a 'Report Bug/Give Feedback' link. Below the header is a navigation bar with tabs for 'WORKFLOWS', 'SETTINGS', and 'ARCHIVAL'. The 'WORKFLOWS' tab is selected. A search bar and filter controls are present above the main table. The table lists workflow details:

| WORKFLOW ID                                     | RUN ID                               | NAME              | STATUS    | START TIME           | END TIME             |
|-------------------------------------------------|--------------------------------------|-------------------|-----------|----------------------|----------------------|
| delete-user-data-auth0 626de950dc765f00706b38f4 | 59aaea84-9c76-4b41-8a12-3b99ebe5ec6e | DeleteAllUserData | Completed | Apr 30, 2022 8:02 PM | Apr 30, 2022 8:03 PM |

Clicking on any workflow run ID will provide additional details about its execution:



We've come a long way in this chapter as we have worked through the steps of adding a subgraph with a single `Mutation` field that can be invoked by a Marked user to trigger a series of mutations in other subgraphs. The `deleteAllUserData` field gives us the best of both worlds where we can design the API schema in a client-focused way and expose a single mutation that will do all of the heavy lifting to coordinate user data deletion across services, but without compromising the separation of concerns between the subgraphs.

From either an architectural or a schema design perspective, there is still an outstanding detail that could be improved upon though. Namely, we've introduced a type of cyclic dependency into our system where the gateway sends an operation containing the `deleteAllUserData` field to the workflows service during query plan execution, and the workflows service calls back to the gateway to execute a series of mutations on other subgraphs before the original operation completes. In a more advanced implementation, one way that we could address this concern is to run a separate instance of the gateway to handle requests from the workflows service while continuing to route requests from front-end clients to the existing gateway.

Beyond the architectural concerns, we may also consider some schema design improvements if we run a separate instance of the gateway to handle requests from the workflows service. Any mutation that we added the `@scope` field to this chapter is effectively off-limits to any developers building front-end client applications because they require a JWT generated for a machine-to-machine application to pass authorization. This may become a source of confusion for client developers, so at a bare minimum, documentation about these fields' intended usage could be improved in the SDL description blocks.

When using managed federation with Apollo Studio, there's a feature called "Contracts" that allows you to selectively include or exclude fields from different variants of a schema by using a special `@tag` type system directive. This feature would allow us to apply `@tag` to our "internal use only" fields (`deleteAccount`, `deleteProfile`, `removeUserFromNetworks`, and `deleteAllUserBookmarks`) and provide a version of the supergraph without these fields for general use to front-end clients while including these fields in another version that is served by the dedicated workflows instance of the gateway. However, this feature is currently only available for enterprise Apollo Studio accounts.

## Summary

In this chapter, we implemented a workflow for handling the cascading deletion of all of a user's data across Auth0 and MongoDB using Temporal as an orchestrator. We added the workflows service and added a single `deleteAllUserData` field to its root `Mutation` type to initiate a workflow that contained four distinct activities. These activities made calls back to the gateway to remove relevant user data. We also added an `@scope` type system directive to authorize the workflows service's requests to the gateway with a special JWT generated for a machine-to-machine application in Auth0.

Now that our subgraph services are complete, in the next chapter we'll upgrade from unmanaged to managed federation to simulate how we would access and maintain the supergraph schema our federated graph in a production environment.

## Chapter 9

# Managed Federation with Apollo Studio

In this chapter, we will:

- Set up a free Apollo Studio account
- Create a deployed graph in Apollo Studio and publish subgraph schemas to it
- Configure the existing gateway to run use managed federation
- Gain insight into operation latency by capturing federated traces and sending that data to Apollo Studio
- Use schema checks in Apollo Studio to validate proposed changes

## Why Use Managed Federation?

We've made amazing strides in building out the Marked GraphQL API over the last eight chapters. It now consists of four distinct subgraphs with all of the queries and mutations required for client developers to build out the essential features of any front-end app for Marked. As we built out these subgraphs, we relied on `IntrospectAndCompose` to poll the subgraphs and recompose the supergraph schema on the fly in the gateway as we worked. This approach worked well enough but—as alluded to in Chapter 1—a production environment will require a different approach.

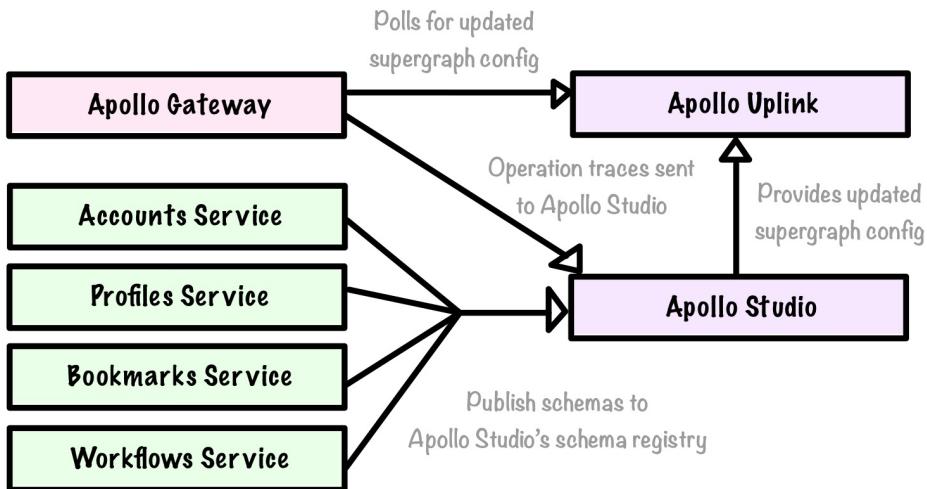
To better understand why, let's pause and consider what a production-ready workflow would look like. GraphQL APIs are meant to be flexible enough to evolve in response to changing product requirements, and federated graphs are no exception. If a client application needs to add a field on the root `Query` type to support a new feature, then the applicable subgraph should be able to make that update and trigger the recomposition of the supergraph schema.

But what happens if a subgraph makes a change that results in a composition failure, such as removing an entity type that other subgraphs have extended with computed fields? This scenario would cause the gateway to throw an error and lead to downtime for the API. Ideally, we would

have an easy way to detect potential issues with composition before the gateway is aware of them so that subgraph service owners can resolve them before deploying their updated code to production.

This is where a schema registry such as [Apollo Studio](#) comes in. A schema registry will act as the source of truth for API's type definitions and help subgraph teams collaborate more fluidly as they update their portions of the schema. And when teams contribute changes to their subgraphs where they unknowingly break composition, the schema registry can also detect these errors in advance and continue to serve the previous composable version of the supergraph schema until further changes are made that allow composition to proceed. Ultimately, a schema registry allows us to maintain separation of concerns within the gateway itself by isolating supergraph composition from query planning and execution.

A version of our distributed GraphQL API architecture that incorporates managed federation will end up looking something like this:

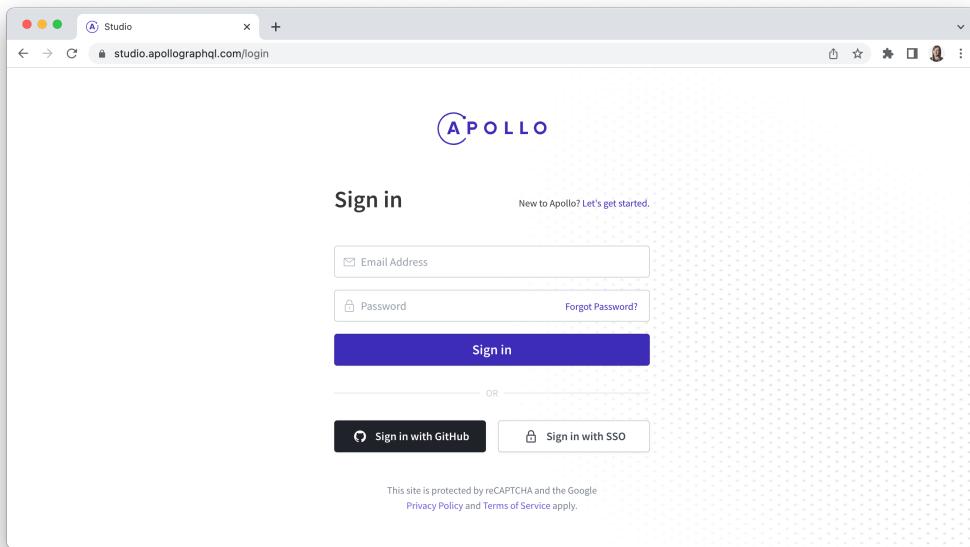


In the sections that follow, we will set up a free Apollo Studio account, publish subgraphs to its schema registry, and reconfigure the existing gateway to use managed federation as pictured in the above diagram.

## Publish Subgraphs to a Deployed Graph in Apollo Studio

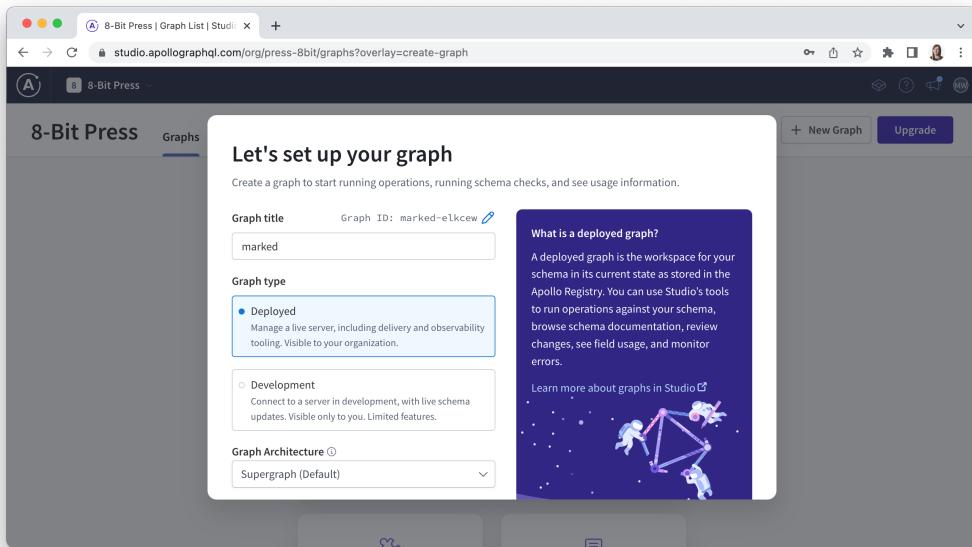
To set up managed federation, you will need an Apollo Studio account. Apollo Studio offers a generous free tier that will allow us to manage the Marked schema in its registry and run millions of queries against our graph every month. You can sign up for an Apollo Studio account by following these quick steps.

First, navigate to <https://studio.apollographql.com/> to register for a new account:

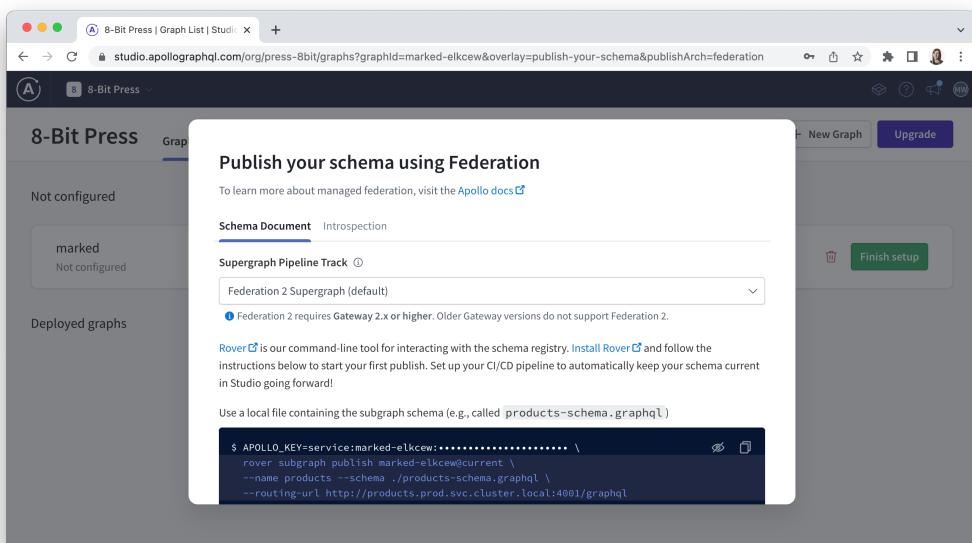


Click the “Create an account” link and then create a new account either using your email or GitHub account. Once your account has been created, you’ll have the option to choose a “Deployed” or “Development” graph. A deployed graph can take advantage of Apollo Studio features such as its schema registry and metrics reporting, and is typically meant for team collaboration. A development graph will poll your development server for schema updates as you work, much like the Sandbox version of Apollo Studio Explorer does but provides an extended feature set compared to Sandbox.

For our purposes, we will choose a deployed graph because we want to publish Marked’s subgraph schemas to the registry. It’s also important to note that a single graph in Apollo Studio will contain multiple subgraphs. In other words, we only need to create one graph for the entire Marked GraphQL API and we will publish all of its subgraph schemas to that graph. Give the graph a unique name, leave the default “Supergraph” option selected in the “Graph Architecture” drop-down, and click the “Next” button below it to proceed:



A new modal will appear with instructions on explaining how to publish subgraph schemas to the Apollo Studio schema registry:



As these instructions suggest, we need to have the Rover CLI installed to publish the subgraph’s schema into the registry. If you haven’t installed Rover yet, then you can refer to the “Required Software” section of the Preface for instructions on how to do this.

Before publishing a subgraph schema, we need to familiarize ourselves with one more important Apollo Studio concept called a *variant*. A variant typically corresponds to some environment where we would run a GraphQL API. For example, a graph might have development and production variants. A graph can have many different variants if needed, which means there may be just as many slightly different versions of a graph’s schema depending on what state the corresponding environments are in, but typically, there will be a high degree of overlap between the variants. Related to variants, a *graph ref* is comprised of the ID of the graph plus a variant name with an @ symbol as a delimiter.

The first subgraph schema that we’ll push to Apollo Studio will be from the accounts service and we’ll use the `rover subgraph publish` command to do so. There are two different ways that we can tell Rover where to find this subgraph’s schema. The first is to point to the `.graphql` file that contains the SDL-based type definitions for the subgraph, and the second is to send an introspection query to the accounts service’s endpoint and pipe the output into the `rover subgraph publish` command. We will choose the second option as we’d quickly discover that we have a composition error in Apollo Studio when simply pointing to the `schema.graphql` file because the definitions for the authorization-related directives aren’t directly available in there (recall that they are concatenated onto the subgraph SDL in `accounts/src/index.js` instead).

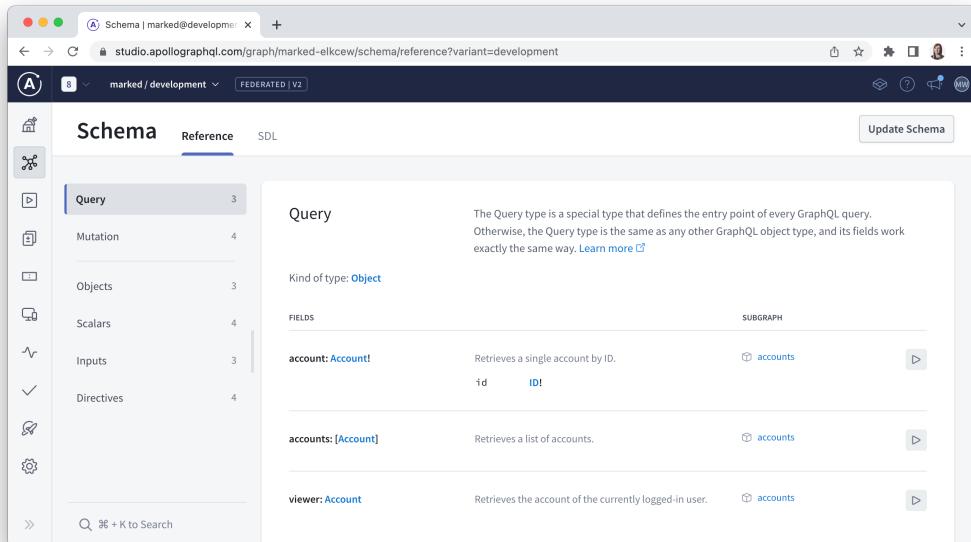
Using the following example command, replace the `APOLLO_KEY` variable and the graph ID portion of the graph ref with the values that correspond to the graph you just created. The directory location will not matter when you run this command:

```
rover subgraph introspect http://localhost:4001 |  
APOLLO_KEY=XXXXXXXXXXXXXX \  
rover subgraph publish <YOUR_GRAPH_ID>@development \  
--name accounts \  
--schema - \  
--routing-url http://localhost:4001
```

If you omit the @ symbol and the variant name and only specify the graph ID as an argument to publish, then Apollo Studio will create a default variant called `current`.

Additionally, instead of using the [Graph API key](#) provided in the “Publish your schema using Federation” modal, you can create a [Personal API key](#) for Apollo Studio and save to a [configuration profile](#) using the `rover config auth` command so that you don’t need to set the `APOLLO_KEY` variable each time you run a Rover command for any graph that you create in your Apollo Studio organization.

Over in Apollo Studio now, if we refresh the browser and navigate to the Schema page and look under the Reference tab, then we should see that the types and fields that are defined in the accounts service are now in the registry:

A screenshot of the Apollo Studio interface. The title bar says "Schema | marked@development". The main navigation bar has tabs for "Schema" (which is selected), "Reference", and "SDL". On the left, there's a sidebar with icons for Query, Mutation, Objects, Scalars, Inputs, Directives, and a search bar. The "Reference" tab is active, showing a table of contents on the left with sections like "Query" (3 items), "Mutation" (4 items), "Objects" (3 items), "Scalars" (4 items), "Inputs" (3 items), and "Directives" (4 items). The main content area is titled "Query" and describes it as a special type for GraphQL queries. It shows three definitions: "account: Account!" which retrieves a single account by ID, "accounts: [Account]" which retrieves a list of accounts, and "viewer: Account" which retrieves the account of the currently logged-in user. Each definition includes a "SUBGRAPH" section with a "accounts" icon.

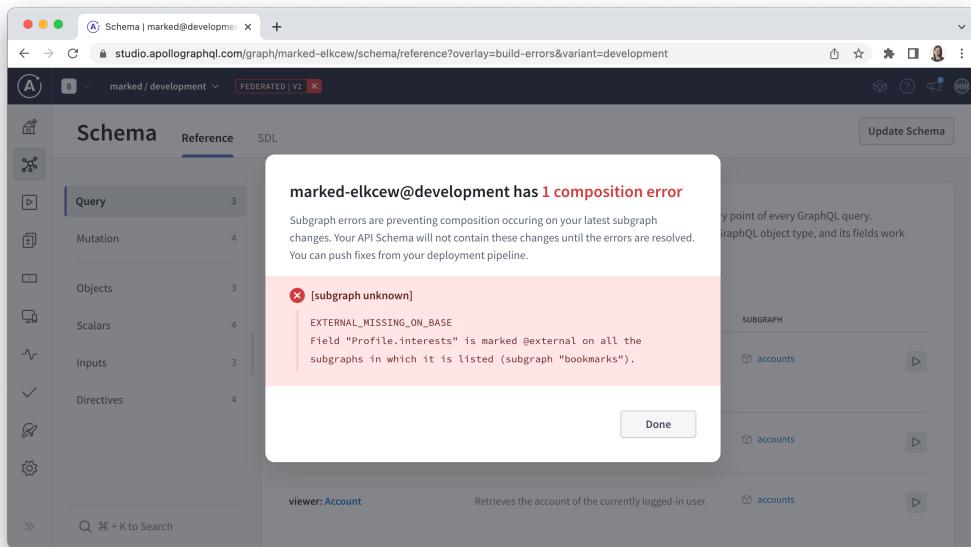
Under the SDL tab, we can also see all three representations of the accounts service's schema as an API schema, supergraph schema, and a subgraph schema. If we proceed with publishing the remaining subgraphs now in the order that we built them, the subgraphs will compose neatly into the supergraph schema in the registry. But let's publish the bookmarks service's schema next instead to see what happens when we publish a new subgraph schema that breaks supergraph composition in Apollo Studio:

```
rover subgraph introspect http://localhost:4003 |
APOLLO_KEY=XXXXXXXXXXXXXX \
rover subgraph publish <YOUR_GRAPH_ID>@development \
--name bookmarks \
--schema - \
--routing-url http://localhost:4003
```

We will see the following error output in the terminal:

```
WARN: The following build errors occurred:
EXTERNAL_MISSING_ON_BASE: Field "Profile.interests" is marked @external on
all the subgraphs in which it is listed (subgraph "bookmarks").
```

Similarly, if we refresh Apollo Studio, then we will see that the “Federated V2” button in the header has turned red. The same error message appears in a modal when we click that button:



This error occurs because we stubbed out the `Profile` entity in the bookmarks service so we could reference and extend it, but we also added the `interests` field to that definition with an `@external` directive applied to support the use of the `@requires` directive for the computed `recommendedBookmarks` field. However, the `@external` directive indicates to the gateway that the `interests` field is referenced by the bookmarks service, but it is not resolvable from that service—or any service at the moment, for that matter—which breaks composition.

What we have seen here is a key feature of managed federation. We attempted to publish a change to the schema registry that breaks composition but Apollo Studio doesn't allow the supergraph schema to be updated until the issue is resolved. That means that a gateway instance running in managed mode will continue to serve the schema exactly as it was when we initially published the accounts service's schema, and Apollo Studio will wait until the composition error is resolved before providing an updated supergraph SDL to the gateway. This is a very important feature to prevent breaking changes from being shipped to a gateway in production.

We can fix this error by publishing the profiles service's schema next:

```
rover subgraph introspect http://localhost:4002 |
APOLLO_KEY=XXXXXXXXXXXXX \
rover subgraph publish <YOUR_GRAPH_ID>@development \
--name profiles \
```

```
--schema - \
--routing-url http://localhost:4002
```

Composition will succeed now that the `interests` field can be resolved by profiles service and we will be able to see all three subgraph schemas in Apollo Studio:

```

API schema
No url set
May 3, 2022 at 9:28 PM MDT • 388c26

1  """An account is unique Auth0 user."""
2  type Account {
3      """The unique ID associated with the account."""
4      id: ID!
5
6      """The date and time the account was created."""
7      createdAt: DateTime!
8
9      """The email associated with the account (must be unique)."""
10     email: String!
11
12     """Metadata about the user that owns the account."""
13     profile: Profile
14 }
15
16 """A bookmark contains content authored by a user."""

```

Lastly, we can publish the workflows service's schema:

```

rover subgraph introspect http://localhost:4004 | 
APOLLO_KEY=XXXXXXXXXXXXX \
rover subgraph publish <YOUR_GRAPH_ID>@development \
--name workflows \
--schema - \
--routing-url http://localhost:4004

```

Before moving on, check the Schema page in Apollo Studio to confirm that all four subgraphs are now composed into the supergraph schema.

## Configure Apollo Gateway to Run in Managed Mode

Now that the schema is fully composed in the Apollo Studio schema registry, we can update the gateway configuration to use managed federation instead. Doing so only requires two updates

to the gateway service's code. The first step is adding the graph API key as a variable named APOLLO\_KEY in its .env file along with an APOLLO\_GRAPH\_REF variable to indicate which graph and variant from the schema registry should be used with this instance of the gateway:

*gateway/.env*

```
APOLLO_KEY=XXXXXXXXXXXXXX  
APOLLO_GRAPH_REF=<YOUR_GRAPH_ID>@development
```

Next, we no longer need to compose the supergraph SDL at runtime in the gateway, so we can remove the supergraphSdl option from it. The updated ApolloGateway constructor will only contain the buildService option now:

*gateway/src/config/apollo.js*

```
// ...  
  
function initGateway(httpServer) {  
  const gateway = new ApolloGateway({  
    buildService({ url }) {  
      return new RemoteGraphQLDataSource({  
        apq: true,  
        url,  
        willSendRequest({ request, context }) {  
          request.http.headers.set(  
            "user",  
            context.user ? JSON.stringify(context.user) : null  
          );  
        }  
      });  
    }  
  });  
  
  // ...  
}  
  
export default initGateway;
```

Upon restarting the gateway, we should see a message logged to the terminal like this:

```
gateway/
```

```
# ...
Apollo usage reporting starting! See your graph at
  https://studio.apollographql.com/graph/marked-elkcew@development/
Gateway ready at http://localhost:4000/
```

With that, the Marked GraphQL API is officially running in managed mode and the gateway will poll Apollo Uplink for supergraph updates and hot-reload those changes in the gateway without requiring it to restart.

Something interesting to note in the logged message above is that by setting up managed federation we have also enabled *usage reporting* to Apollo Studio. Usage reporting means that the gateway will send [federated trace data](#) to Apollo Studio so we can understand how the API is being used and where potential performance issues may exist. Tracing data also helps power another important feature in Apollo Studio that we will explore in the next section called “Schema Checks.”

To see how this tracing data is put to use in Apollo Studio, we’ll have to run a few operations against our API. However, rather than using the Sandbox version of Explorer, this time we will use the Explorer interface that’s built into Apollo Studio for use with deployed or development graphs. It can be accessed using the same icon in the lefthand menu while logged into Apollo Studio.

Before running any operations, let’s set some headers first. In the “Headers” panel below the editor, add the `apollographql-client-name` header with a value of `Apollo Studio Explorer` and an `apollographql-client-version` of `1.0`. These headers will help enrich our traces with client-specific metadata so we can understand how the API is being used and by whom as well. Now add the following operation document and variables to the Explorer editor:

#### GraphQL Query

```
query SearchBookmarks($query: String!, $first: Int) {
  searchBookmarks(query: $query, first: $first) {
    edges {
      node {
        title
        url
      }
    }
  }
}
```

### Query Variables

```
{  
  "query": "graphql",  
  "first": 10  
}
```

When you run the operation for the first time you will be prompted to set an endpoint for the API so add <http://localhost:4000> to the connection settings, click the “Save” button in the modal, and then run the operation several times:

The screenshot shows the Apollo Studio interface with the following details:

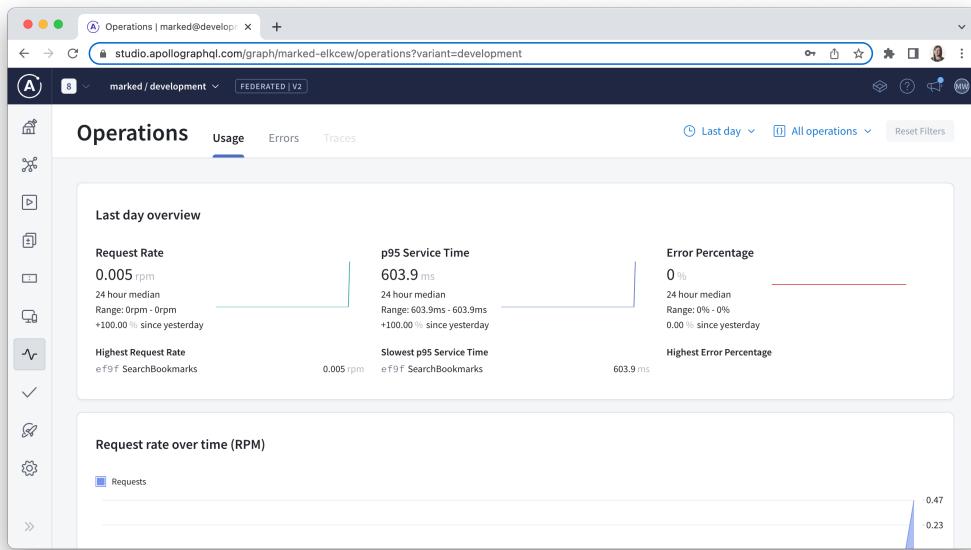
- Left Sidebar (Documentation):** Shows the project structure with a **Query** operation selected.
- Middle Panel (Operation):** Displays the GraphQL query:

```
query SearchBookmarks(  
  $query: String!  
  $first: Int  
) {  
  searchBookmarks(query: $query, first: $first) {  
    edges {  
      node {  
        title  
        url  
      }  
    }  
  }  
}
```
- Bottom Left (Variables):** Shows the variables for the query:

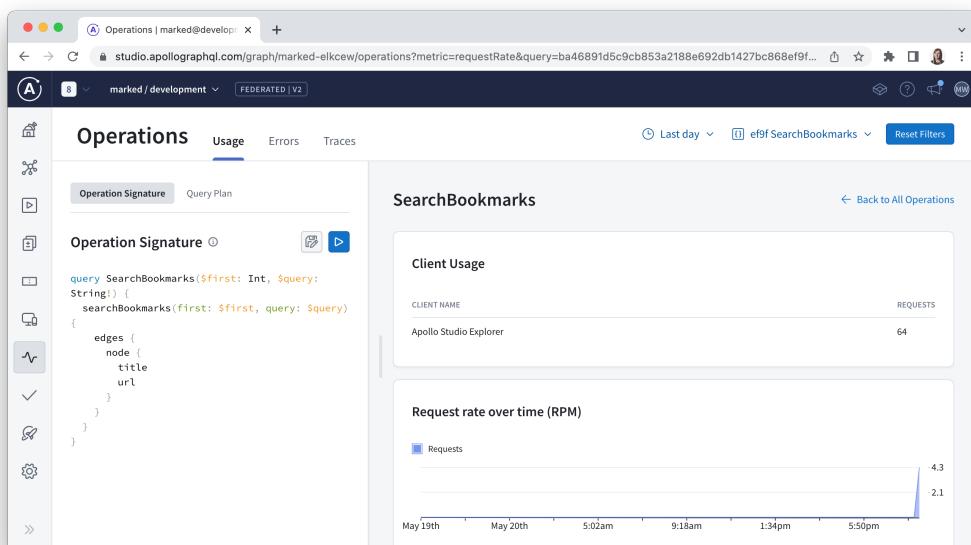
```
1  "first": 10,  
2  "query": "graphql"  
3  
4
```
- Bottom Right (Response):** Shows the JSON response from the API call, indicating a successful 200 status code with a response time of 707ms and 417B size.

```
"data": {  
  "searchBookmarks": {  
    "edges": [  
      {  
        "node": {  
          "title": "GraphQL Specification",  
          "url": "https://spec.graphql.org/"  
        }  
      },  
      {  
        "node": {  
          "title": "Life of a GraphQL Query - Validation",  
          "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query-validation-18a8fb52f1898"  
        }  
      },  
      {  
        "node": {  
          "title": "Life of a GraphQL Query - Lexing/Parsing",  
          "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query-lexing-parsing-18a8fb52f1898"  
        }  
      }  
    ]  
  }  
}
```

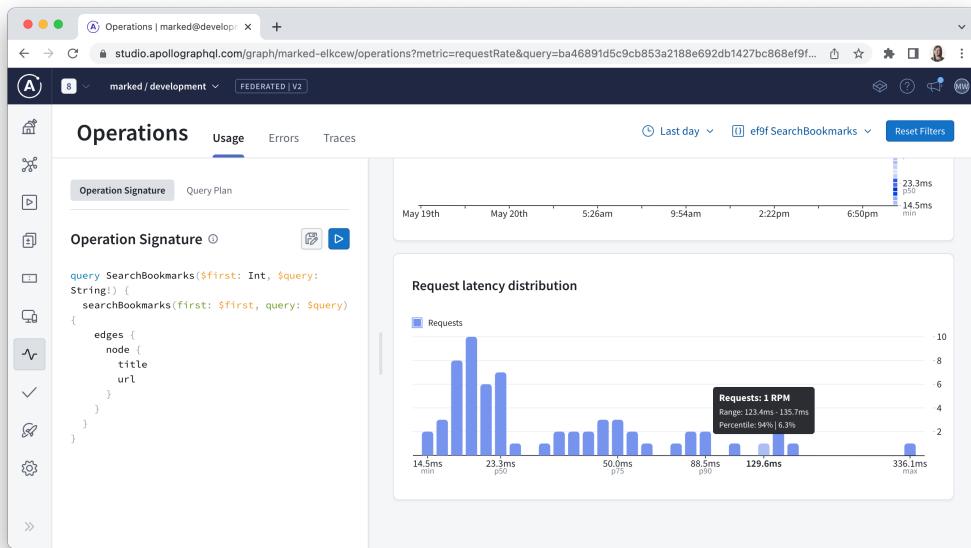
We can jump over to the operations tab now and see the `SearchBookmarks` query listed in the “Last day overview” section:



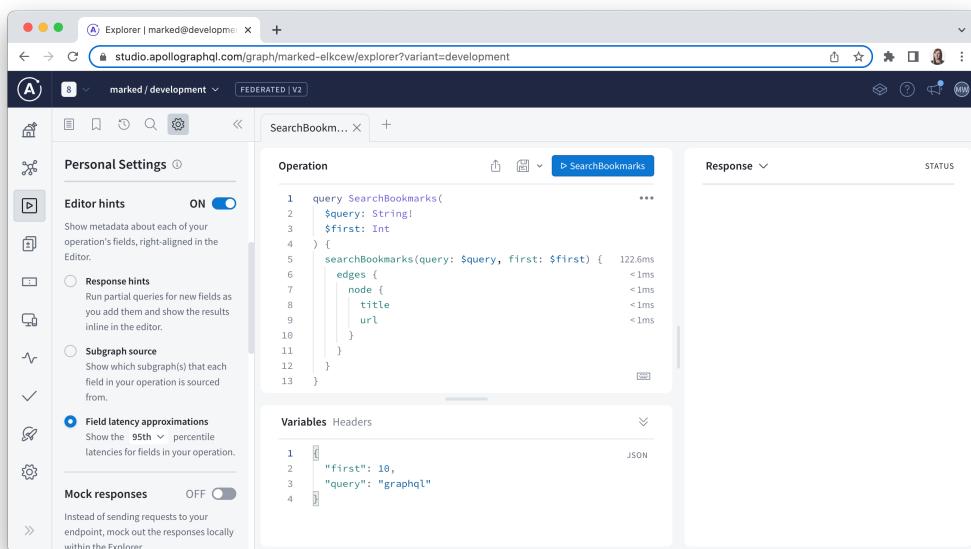
If we click on that operation, then we can see that the named “Apollo Studio Explorer” client has run the operation a number of times:



Similarly, if we scroll down the page we will see the request latency distribution:



Back in Explorer, we can get more granular insights into how various fields contribute to request latency by taking advantage of some of the advanced settings here and turning on editor hints:



A key thing to note about using the full-featured version of Explorer with a deployed graph is that Apollo Studio will not automatically poll a development environment for updates—we would need to publish a subgraph schema again if we made any changes to it. Development graphs, however, can poll a local endpoint for updates while providing access to a broader range of features than the Sandbox version of Explorer does.

## Safely Evolve Subgraphs with Schema Checks

Now that all of the subgraph schemas have been published to the registry, the gateway is running in managed mode, and we are reporting usage data to Apollo Studio, let's imagine what would happen when the Marked app is deployed to production one day. First, we would want to create a separate variant in Apollo Studio for the production version of the schema. Most commonly, subgraph schema publication would be automated during some kind of deployment workflow. Doing so would allow us to continue to work on the development variant without affecting the version of the schema that's exposed to clients in production environments.

Now let's imagine that the Marked application has been serving client traffic for some time and we now need to make a change to the schema. This change may be necessary due to a new product requirement or a change in one of the underlying data sources that back a subgraph. GraphQL APIs are meant to evolve over time by design, but how can we be sure that the proposed updates won't cause breaking changes for clients that may depend on the affected types and fields?

This is where *schema checks* come in. Schema checks are another feature of Apollo Studio (recently made free for any account type) that allow you to statically analyze proposed changes to a subgraph schema against a slice of client traffic for a specific time range. The schema check will flag any potentially breaking changes based on this traffic and it will also allow us to identify any new compositions issues before the subgraph schema is republished.

To try out the schema check process, let's begin by making what we would expect to be a non-breaking change to the bookmarks service's schema. Specifically, we will set the `title` field's output type to be a non-nullable `String` now:

`bookmarks/src/graphql/schema.graphql`

```
# ...

"""
A bookmark contains content authored by a user.

"""

type Bookmark {
    # ...
    "A title to describe the bookmarked content."
    title: String!
    "The URL of the page to be bookmarked."
```

```
    url: URL!
}

# ...
```

We expect this change to be non-breaking because any clients that depend on this field can still expect it to be there. In fact, now that the field is non-nullable these clients can be even more certain that a `title` value will be available when requested.

Without publishing this change to the Apollo Studio registry yet, let's run a schema check using the `rover subgraph check` command to verify that this change is safe:

```
rover subgraph introspect http://localhost:4003 |
APOLLO_KEY=XXXXXXXXXXXXXX \
rover subgraph check <YOUR_GRAPH_ID>@development \
--name bookmarks \
--schema -
```

We should see the following output in the terminal, indicating that the proposed change to the `bookmarks` service's schema is safe:

```
Checking the proposed schema for subgraph bookmarks against marked-elkcew@development
Check Result:

Compared 1 schema changes against 1 operations


Change	Code	Description
PASS	FIELD_CHANGED_TYPE	field `Bookmark.title`: type `String` changed to `String!`


View full details at https://studio.apollographql.com/graph/marketed-kcew/operationsCheck/602fb8e-4c09-4b66-af0c-e837ed62c2b5?variant=development
```

You can see the [full list of types of schema changes](#) in the Apollo Studio documentation.

We can also see a record of the outcome of this schema check on the Checks page in Apollo Studio:

| BRANCH                                                    | TASKS               | VARIANT     | AUTHOR     | COMMIT ID |
|-----------------------------------------------------------|---------------------|-------------|------------|-----------|
| PASSED router<br>bookmarks • Initiated a few seconds ago. | Build<br>Operations | development | Mandi Wise | 22794a4   |

Now let's try running a check against a potentially breaking change by changing the output type of the non-null `url` field to nullable so that there is no longer a guarantee to clients who have queried this field that a value will be available:

`bookmarks/src/graphql/schema.graphql`

```
# ...

"""
A bookmark contains content authored by a user.

"""

type Bookmark {
    ...
    "A title to describe the bookmarked content."
    title: String!
    "The URL of the page to be bookmarked."
    url: URL
}

# ...
```

Next, we'll rerun the schema check:

```

rover subgraph introspect http://localhost:4003 |
APOLLO_KEY=XXXXXXXXXXXXXX \
rover subgraph check <YOUR_GRAPH_ID>@development \
--name bookmarks \
--schema -

```

And we should see the following output this time:

Checking the proposed schema for subgraph bookmarks against marked-elkcew@development  
Compared 2 schema changes against 1 operations

| Change | Code               | Description                                                |
|--------|--------------------|------------------------------------------------------------|
| PASS   | FIELD_CHANGED_TYPE | field `Bookmark.title`: type `String` changed to `String!` |
| FAIL   | FIELD_CHANGED_TYPE | field `Bookmark.url`: type `URL` changed to `URL`          |

View full details at <https://studio.apollographql.com/graph/marked-elkcew@development/operationsCheck/ff269484-6c0c-4919-3d05461c9b5f?variant=development>  
error[E030]: This operation check has encountered 1 schema change that would break operations from existing client traffic.

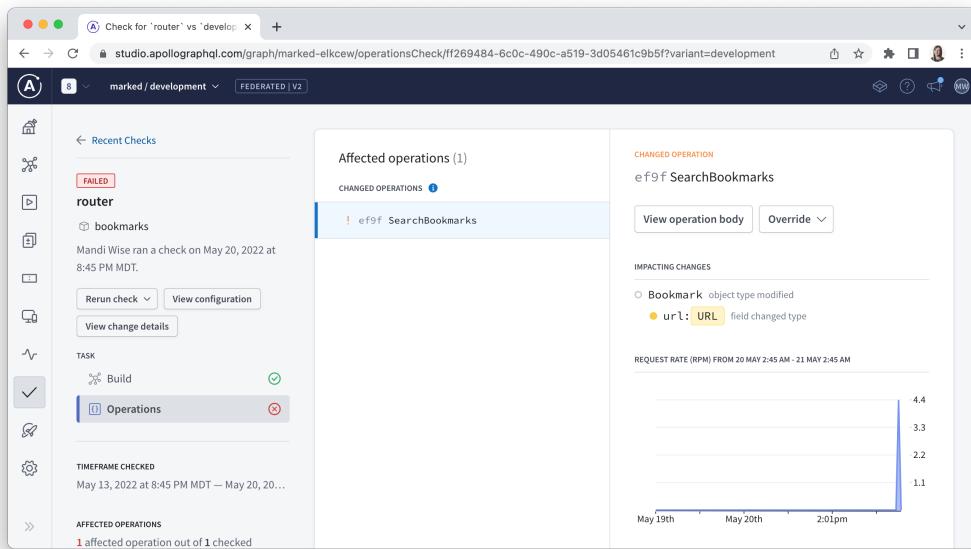
The changes in the schema you proposed are incompatible with graph marked-elkcew@development. See <https://www.apollographql.com/docs/studio/schema-checks/> for more information on resolving operation check errors.

Over in Apollo Studio, we can also see the failed check:

The screenshot shows the Apollo Studio interface with the URL <https://studio.apollographql.com/graph/marked-elkcew/checks?tab=list&variant=development>. The 'Recent Checks' section displays two entries:

- FAILED** router: bookmark • Initiated 3 minutes ago. Tasks: Build (green), Operations (red). Variant: development. Author: Mandi Wise. Commit ID: 22794a4.
- PASSED** router: bookmark • Initiated 4 minutes ago. Tasks: Build (green), Operations (green). Variant: development. Author: Mandi Wise. Commit ID: 22794a4.

If we click on the check, we can see detailed information about why the check failed:



Additionally, if we scroll down this page, then we can also see what clients are affected by this change. This insight can help us make an informed decision about whether we want to override this potential breaking change if flagged again in the future or if we need to follow up with the identified clients before publishing this change to the schema.

## Summary

In this chapter, we explored the use case for managed federation, particularly for production environments where it's important to validate subgraph schema changes in advance to ensure that they won't break composition of the supergraph for a running gateway. We then published the accounts, bookmarks, profiles, and workflows subgraphs to the Apollo Studio schema registry and configured the existing Apollo Gateway to use managed federation automatically by adding APOLLO\_KEY and APOLLO\_GRAPH\_REF environment variables to it.

We also took advantage of some additional features of Apollo Studio that are enabled by the federated traces that the gateway sends to it via its usage reporting mechanism. Lastly, we used schema checks to determine that proposed changes to a subgraph schema will be safe for clients that are actively querying the API and will result in a valid supergraph composition.

In the final chapter, we'll wrap up our survey of advanced GraphQL techniques by trying out the Rust-based Apollo Router, which is a newer addition to Apollo's suite of open-source software.

# Apollo Router

In this chapter, we will:

- Configure Apollo Router and run it in a Docker container
- Replace Apollo Gateway with Apollo Router while using managed federation
- Add a proxy server in front of the router to handle authentication

## Configure Apollo Router

[Apollo Router](#) is the new Rust-based graph routing runtime that can be used with Apollo Federation. Rust is a lower-level language than JavaScript so the router has some notable performance gains over Apollo Gateway. The router is also highly configurable via a user-friendly YAML file, so for many use cases, it's not necessary to write any Rust to use the router as a replacement for the gateway. For advanced implementations, it is also possible to write a [native Rust plugin](#) for the router as well.

When planning any migration from an Apollo Gateway to Apollo Router, it's important to take stock of what customizations were made to the existing gateway. Throughout this book, we ended up making three key configurations to the gateway. In Chapter 2, we extracted a validated JWT from the `req.user` object in Apollo Server's context and then used a `RemoteGraphQLDataSource` in the gateway to forward that validated token to the subgraphs. Later on in Chapter 7, we added a depth-limiting validation rule and used the `formatError` option in Apollo Server to mask errors that may reveal too much information about the structure of the API schema or underlying data sources in production environments.

In this chapter, we will focus on configuring the router to pass validated JWTs to subgraphs but we will hold off on replicating depth limiting or doing any special error formatting here as these considerations would currently require writing custom Rust plugins. We're also going to run Apollo Router using Docker, just as we did with the Temporal server. To simplify the command that we'll use to start up this Docker container, we'll create a small `docker-compose.yaml` file to store the container configuration options that we'll need to run the router.

## Some Essential Docker Concepts

If you're new to Docker, then there are a few key concepts we need to define before we proceed with running Apollo Router with Docker.

An *image* allows us to package up an application's code with all of the specific dependencies required to run it. As a result, images make it easy for us to reliably move an app from machine to machine (for example, from a MacBook to a Linux machine on a cloud host). A key feature of images is that they are immutable because they represent what an app and its supporting environment look like at a particular point in time.

We then use images to create *containers*. In other words, a container is a running instance of a given image. Reciprocally, an image is essentially a snapshot of a container that can be used to recreate other identical containers. Unlike images, running containers can be modified, so that means we can create a container for a database and then write data to it.

[Docker Compose](#) helps streamline the process of defining and running multiple containers for an application. Using a YAML file we can declare all of the different services required by an app, what image to use for each service, and many more configuration options. Once assembled, a Compose file can be used to start all of the containers with a single command. We'll only need one container to run Apollo Router but we could add more to its `docker-compose.yaml` file in the future if needed.

To learn more about these key concepts, see the [Docker official documentation](#).

Create a `router` subdirectory in the root project directory and add a `docker-compose.yaml` file to it with the following code:

`router/docker-compose.yaml`

```
services:
  apollo-router:
    image: ghcr.io/apollographql/router:v0.9.2
    container_name: apollo-router
    env_file:
      - ./env
    volumes:
      - ./router.yaml:/dist/config/router.yaml
    ports:
      - "5000:5000"
```

In the code above we define an `apollo-router` service with some specific configuration options:

- **image:** This is the name of the image used to create the container. In this case, we use an [official router image from Apollo](#).

- **container\_name**: We provide a unique name to use for the container, rather than using the default container name assigned by Docker.
- **env\_file**: There are a few ways that we can set environment variables in a container and the option that we've chosen is to provide an entire `.env` file to the container that we'll create in the `router` directory shortly.
- **volumes**: We mount host paths (located on our computer) into a container using the `volumes` key. When we build a Docker image we end up with a writable container layer on top, but Docker containers can be created and deleted on the fly and we will often want to keep any data created by the container from build to build. Volumes allow us to persist the data generated and used by Docker containers on the host system so it may be used with subsequent builds (and potentially by other services). To customize the router, we'll need to create a `router.yaml` file on the host machine and then mount it in the container when the router is running so it can pick up these options.
- **ports**: We use the `ports` key to set up port forwarding so that the ports these services use inside of Docker are also available on the host machine. Setting up port forwarding for the router means that we will be able to access it on our computer using `localhost` just as we have with the gateway.

Given that we referenced `.env` and `router.yaml` files in the `docker-compose.yaml` file, we should create those files before we try to create the router container. Apollo Router supports managed federation too, so we'll need to provide it with the `APOLLO_KEY` and `APOLLO_GRAPH_REF` variables just as we did the gateway. We'll also add the `APOLLO_ROUTER_HOT_RELOAD` variable to restart the router automatically when changes are made to the `router.yaml` configuration file:

`router/.env`

```
APOLLO_KEY=XXXXXXXXXXXX  
APOLLO_GRAPH_REF=<YOUR_GRAPH_ID>@development  
APOLLO_ROUTER_HOT_RELOAD=true
```

Apollo Router supports unmanaged federation as well, but it doesn't have any configuration options analogous to using `IntrospectAndCompose` with the `supergraphSdl` option in the gateway. Instead, you will need to generate the supergraph SDL using the Rover CLI and then pass a `.graphql` file containing that SDL to the router on start-up when using unmanaged federation with the router.

Next, we'll create the `router.yaml` file in the `router` directory with a few options that will allow the router to run in a Docker container while still making requests to `localhost` to send operations to the running subgraphs during query plan execution:

*router/router.yaml*

```
server:  
  listen: 0.0.0.0:5000  
  override_subgraph_url:  
    accounts: http://host.docker.internal:4001  
    profiles: http://host.docker.internal:4002  
    bookmarks: http://host.docker.internal:4003  
    workflows: http://host.docker.internal:4004
```

The router will listen on port 5000 in the container on the all interfaces address inside of Docker. Additionally, we can't use `localhost` for the subgraph endpoint because `localhost` will mean something different inside a Docker container than it does on the host machine, so we use the router's `override_subgraph_url` configuration option to set each subgraph endpoint using the `http://host.docker.internal` URL instead so that Docker knows to send requests from the router to the host machine.

With our configuration files in place, we can start up the router with managed federation:

*router/*

```
docker compose up
```

If we head over to `http://localhost:5000` in a browser, then we will see the familiar landing page for the Sandbox version of Explorer. Let's try querying some publicly available data first:

*GraphQL Query*

```
query SearchBookmarks($query: String!, $first: Int) {  
  searchBookmarks(query: $query, first: $first) {  
    edges {  
      node {  
        title  
        url  
      }  
    }  
  }  
}
```

*Query Variables*

```
{  
  "query": "graphql",
```

```
    "first": 2
}
```

#### API Response

```
{
  "data": {
    "searchBookmarks": {
      "edges": [
        {
          "node": {
            "title": "GraphQL Specification",
            "url": "https://spec.graphql.org/"
          }
        },
        {
          "node": {
            "title": "Life of a GraphQL Query – Validation",
            "url": "https://medium.com/@cjoudrey/life-of-a-graphql-query-validation-18a8fb52f1898"
          }
        }
      ]
    }
  }
}
```

As we can see, Apollo Router resolves the query just like Apollo Gateway does and it only took a small amount of configuration code to drop it in as a replacement. But what happens when we try an operation that returns data that can only be viewed by an authenticated user? Add a valid JWT as an `Authorization` header in Explorer and try running the following query now:

#### GraphQL Query

```
query SearchProfiles($query: String!, $first: Int) {
  searchProfiles(query: $query, first: $first) {
    edges {
      node {
        username
        account {
          email
        }
      }
    }
  }
}
```

```
        }
    }
}
```

### Query Variables

```
{
  "query": "mark",
  "first": 2
}
```

The initial response that we see will be a server error that occurs because we have not yet configured CORS for the router so that it allows the `Authorization` header. We could fix that issue by adding the following code to the `router.yaml` file:

`router/router.yaml`

```
server:
  listen: 0.0.0.0:5000
  cors:
    origins:
      - https://studio.apollographql.com
    allow_headers: [Content-Type, Authorization]
# ...
```

However, even with this code in place, we will still see an error when we rerun the query. This time it will be a `Not authorized!` error and it occurs because we're not validating the JWT in the router or setting it on `user` header in the requests to the subgraphs so that subgraphs may in turn use the validated token for field-level authorization.

There are two possible solutions to this problem. The first would be writing [a custom plugin in Rust](#) that can handle JWT validation for Auth0's asymmetric signing algorithm and then add it to the router (just like the `jwks-rsa` middleware does in the gateway). Writing Rust is outside the scope of this book so we'll opt for the alternative approach, which is to set up a Node.js-based proxy server in front of the router to validate tokens before forwarding them to this service.

### Optional: Additional Helpful Docker Commands

Now that Docker is up and running locally, you may find the following additional commands helpful for managing your containers.

Run Docker containers in the background (detached mode):

```
docker compose up -d
```

Recreate containers before running, even if their configuration and image haven't changed:

```
docker compose up --force-recreate
```

Stop all of your running containers:

```
docker container stop $(docker container ls -a -q)
```

List all of the images cached on your system:

```
docker image ls
```

Remove unused images:

```
docker image prune
```

Check container memory usage:

```
docker stats
```

And if something goes sideways and you need to invoke a nuclear option to completely clean up and rebuild the containers and volumes, then run this command:

```
docker container stop $(docker container ls -a -q) && \
docker system prune -a -f --volumes && \
docker compose up
```

## Set Up a Proxy to Handle Authentication

To complete the migration from the gateway to the router, we'll need to set up a lightweight proxy server using Express to validate JWTs before forwarding API requests to the router. We'll be able to repurpose some of the existing code from the gateway, so what we're about to write should look familiar. To begin, we'll create a new auth-proxy subdirectory in the root project directory and then create a package.json file in it:

```
auth-proxy/
```

```
npm init --yes
```

We'll install `dotenv`, `express`, and the same packages that we used to handle JWTs as before. We will also need to install the `cors` and `http-proxy-middleware` packages to use as middleware in the Express server:

*auth-proxy/*

```
npm i cors@2.8.5 dotenv@16.0.0 express@4.17.3 express-jwt@6.1.1  
http-proxy-middleware@2.0.6 jwks-rsa@2.0.5
```

And again, we'll use `nodemon` as a development dependency to restart the server as files change:

*auth-proxy/*

```
npm i -D nodemon@2.0.15
```

In the package.json file, we'll set the `type` key to `module` and also add a `dev` script:

*auth-proxy/package.json*

```
{  
  // ...  
  "type": "module",  
  "scripts": {  
    "dev": "nodemon -r dotenv/config -e env,js ./src/index.js"  
  },  
  // ...  
}
```

We will need a `.env` file for this service as well and it will reuse several variables from the gateway but also include a `ROUTER_ENDPOINT` variable that we will use when configuring the proxy:

*workflows/.env*

```
AUTH0_AUDIENCE=http://localhost:4000/  
AUTH0_ISSUER=https://markedapp.us.auth0.com/  
  
ROUTER_ENDPOINT=http://localhost:5000  
  
NODE_ENV=development  
PORT=4000
```

Again, make sure that the `AUTH0_ISSUER` value is set to the domain of your Auth0 tenant. To finish scaffolding this service, add a `src` directory inside of `auth-proxy` with an `index.js` file in it. The current file structure in `auth-proxy` will look like this:

```
auth-proxy
├── node_modules/
│   └── ...
├── src/
│   └── index.js
├── .env
└── package.json
    └── package-lock.json
```

Now we can set up the token-validating middleware in this service. To do this, we have to create an Express app first and then add the `cors` middleware to it with the Apollo Studio URL as an allowed origin so we can continue to use Explorer to send requests to the API via the proxy. We will also copy and paste over the same JWT-handling middleware that was used in the gateway:

`auth-proxy/src/index.js`

```
import cors from "cors";
import express from "express";
import jwt from "express-jwt";
import jwksClient from "jwks-rsa";

const port = process.env.PORT;
const app = express();

app.use(
  cors({
    origin: ["https://studio.apollographql.com"]
  })
);

const jwtCheck = jwt({
  secret: jwksClient.expressJwtSecret({
    cache: true,
    rateLimit: true,
    jwksRequestsPerMinute: 5,
    jwksUri: `${process.env.AUTH0_ISSUER}.well-known/jwks.json`
  }),
  audience: process.env.AUTH0_AUDIENCE,
  issuer: process.env.AUTH0_ISSUER,
  algorithms: ["RS256"],
  credentialsRequired: false
});
```

```
app.use(jwtCheck, (err, req, res, next) => {
  if (err.code === "invalid_token") {
    return next();
  }
  return next(err);
});
```

Now we can set up the proxy middleware. After the JWT middleware is applied, we'll add the `createProxyMiddleware` by setting the router's endpoint as the target, changing the origin, and hooking into the `onProxyReq` event to set a `user` header to the stringified value that the JWT middleware previously set for the `req.user` object:

*auth-proxy/src/index.js*

```
import { createProxyMiddleware } from "http-proxy-middleware";
// ...

app.use(
  createProxyMiddleware({
    target: process.env.ROUTER_ENDPOINT,
    changeOrigin: true,
    onProxyReq(proxyReq, req) {
      proxyReq.setHeader(
        "user",
        req.user ? JSON.stringify(req.user) : null
      );
    }
  })
);
```

Just as we discussed in Chapter 2 for the requests that Apollo Gateway sends to subgraphs, validating a token and passing it along to other services in this manner would necessitate that communication between the proxy and the router is secure and that the router isn't directly accessible to the outside world. Lastly, we can start up the Express server on port 4000:

*auth-proxy/src/index.js*

```
// ...
app.listen(port, () => {
```

```
    console.log(`Authentication proxy ready at http://localhost:${port}`);  
});
```

Before we can test out the new proxy server with the router, we need to make a final configuration change to the `router.yaml` file. Just as the gateway forwarded the `user` header to the subgraphs, the router will need to do the same. We can do this without writing any Rust by setting the `headers` option in `router.yaml` and then indicate that we want to propagate the `user` header from the incoming request from the proxy server to the subgraphs during query plan execution:

*router/router.yaml*

```
server:  
  listen: 0.0.0.0:5000  
headers:  
  all:  
    - propagate:  
        named: "user"  
override_subgraph_url:  
  accounts: http://host.docker.internal:4001  
  profiles: http://host.docker.internal:4002  
  bookmarks: http://host.docker.internal:4003  
  workflows: http://host.docker.internal:4004
```

Make sure the router is running on <http://localhost:5000> and then start up the proxy:

*auth-proxy/*

```
npm run dev
```

Now we can rerun the `SearchProfiles` query and validate that the router will now authorize and resolve the operation. Note that you will need to change the Explorer endpoint back to <http://localhost:4000> to send the request through the proxy:

*GraphQL Query*

```
query SearchProfiles($query: String!, $first: Int) {  
  searchProfiles(query: $query, first: $first) {  
    edges {  
      node {  
        username  
        account {  
          email
```

```
        }
    }
}
}
```

#### Query Variables

```
{
  "query": "mark",
  "first": 2
}
```

#### API Response

```
{
  "data": {
    "searchProfiles": {
      "edges": [
        {
          "node": {
            "username": "funkybunch",
            "account": {
              "email": "markymarked@markedmail.com"
            }
          }
        },
        {
          "node": {
            "username": "marksalot",
            "account": {
              "email": "marksalot@markedmail.com"
            }
          }
        }
      ]
    }
  }
}
```

The request is now authorized and we receive the data that we expect in the response.

## Summary

Congratulations! It's been a long journey from zero lines of code in the first chapter to a distributed GraphQL API powered by four different subgraphs, managed federation, and Apollo Router, but you made it all the way to the end. We began this chapter by configuring Apollo Router to run with Docker in a local development environment. We then added a Node.js-based proxy server to handle authentication in front of the router. After that, we were able to begin sending requests to the Marked GraphQL API exactly as we did before with Apollo Gateway, but now leveraging Apollo's next-gen graph routing runtime.

# About the Author

Mandi Wise has worked in developer education for over seven years with a focus on full-stack web application development using JavaScript and GraphQL. During that time, she wrote curricula for and instructed numerous full-time and part-time courses and also developed software to support program delivery. She currently leads the Solutions Architecture team at Apollo Graph Inc.

Mandi first discovered her love for building things for the web 20 years ago while spending countless hours handcrafting beautiful table-based websites with spacer GIFs on GeoCities. She currently shares that enduring passion for the web by speaking at numerous developer conferences and meet-ups and actively volunteering in her local tech community.

# **Changelog**

## **v2.0.0**

- Initial release of second edition