neo4j

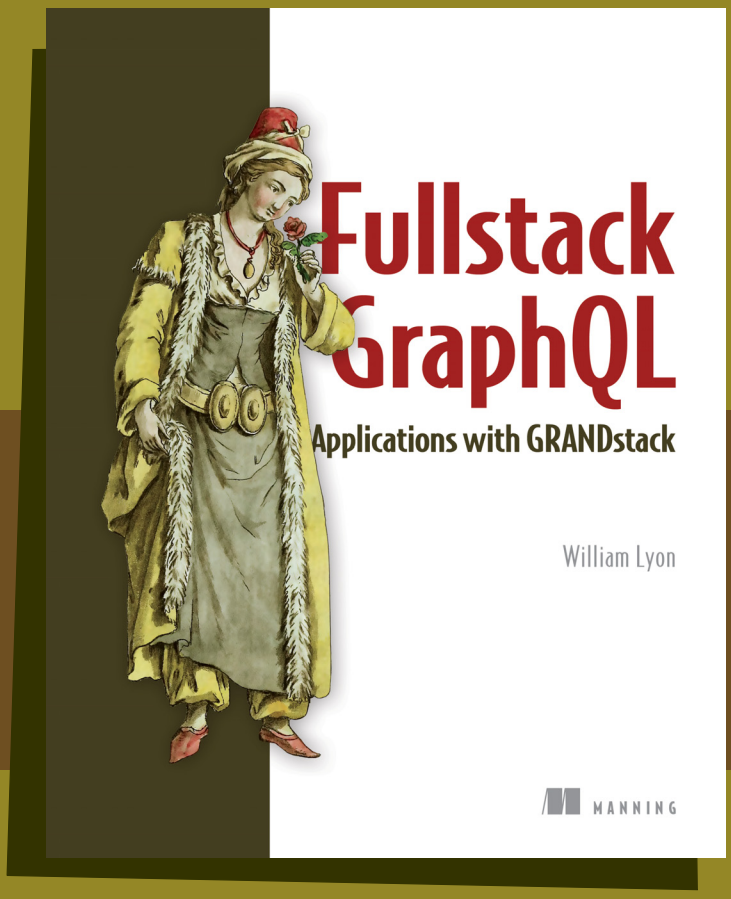
Fullstack GraphQL

Applications with GRANDstack

Essential Excerpts

William Lyon

MANNING



Fullstack GraphQL Applications with GRANDstack

by William Lyon

ISBN 9781617297038
300 pages (estimated)
$39.99

Fullstack GraphQL Applications with GRANDStack

Essential Excerpts

Chapters chosen by William Lyon

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Marija Tudor

ISBN: 9781617298745

contents

foreword

Thank you for downloading this special edition of *Fullstack GraphQL: Building Applications with GRANDstack*. This book is intended for developers interested in building fullstack applications with GraphQL. The successful reader will have some basic familiarity with Node.js and a basic understanding of client-side JavaScript, but most importantly, will have a motivation for understanding how to build GraphQL services and applications leveraging GraphQL.

The goal of this book is to demonstrate how GraphQL, React, Apollo, and Neo4j Database can be used together to build complex, data-intensive fullstack applications. You may be wondering why we've chosen this specific combination of technologies. As you read through the book, I hope you'll realize the developer productivity, performance and intuitive benefits of using a graph data model throughout the stack—from the database to the API—and all the way through the front-end client data fetching code.

The full book is divided into three sections, which together show how to build a fullstack business-reviews application with features such as search and personalized recommendations. The first section focuses on server-side GraphQL with Neo4j, building API functionality that includes not just simple data fetching, but also implementing custom logic using graph database traversals. The second section focuses on building a front-end using React and Apollo Client. And the final section explores more real-world concerns such as authorization, deployment, and rate limiting.

The special edition ebook you are reading now focuses on the first section: implementing a server-side GraphQL API backed by the Neo4j graph database. After reading these selected chapters and working through the exercises, you should have an understanding of how to build APIs with Neo4j and GraphQL, using the GraphQL integration for Neo4j known as neo4j-graphql.js.

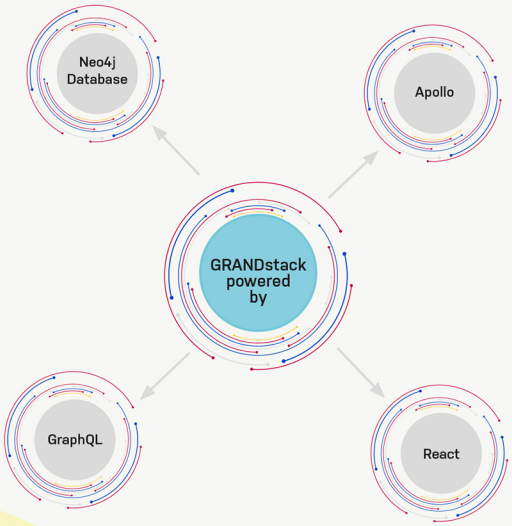
You can find all the relevant code and exercise solutions in the book's GitHub page: github.com/johnymontana/fullstack-graphql-book. Thanks for reading, and I hope you enjoy your fullstack GraphQL journey!

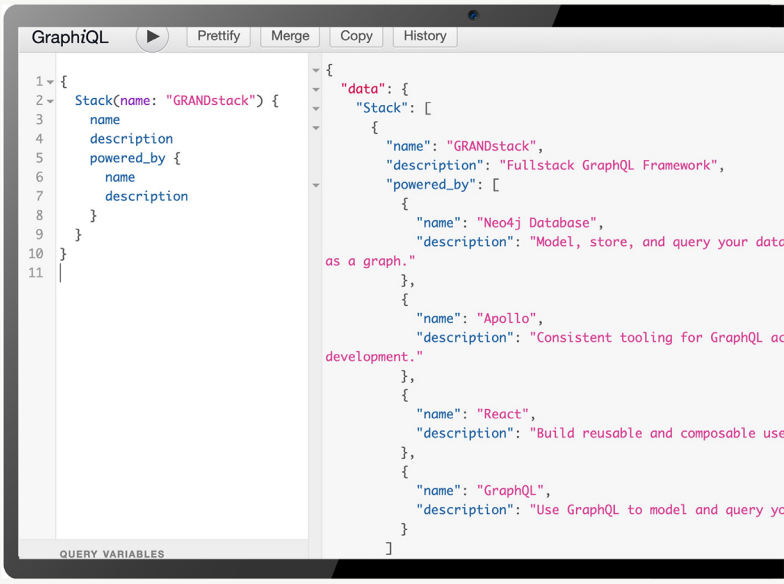
Cheers,
William Lyon
@lyonwj

neo4j

GRANDstack

BUILD FULL-STACK GRAPHQL APPLICATIONS WITH EASE





With GraphQL, you model your business domain as a graph. Build your application with **graphs all the way** down by leveraging the world's leading graph database, Neo4j.

Try out both together in the cloud with Neo4j Sandbox.

GRANDstack.io/sandbox

1 What's the GRANDstack?

This chapter covers

- Understanding the GRANDstack
- Reviewing each technology in the stack
- Looking at the full-stack application we'll build

In this chapter we look at the technologies we'll use throughout the book, specifically:

- **G**raphQL: For building our API
- **R**eact: For building our user interface and JavaScript client web application
- **A**pollo: Tools for working with GraphQL on both the server and client
- **N**eo4j **D**atabase: The database we'll use for storing and manipulating our application data

Together these technologies make up the **GRAND**stack, a full-stack framework for building applications using GraphQL (figure 1.1).

Throughout the course of this book we'll use GRANDstack to build a simple business review application, working through each technology component as we implement it in the context of our application. In the last section of this chapter we review the basic requirements of the application we'll build throughout the book.

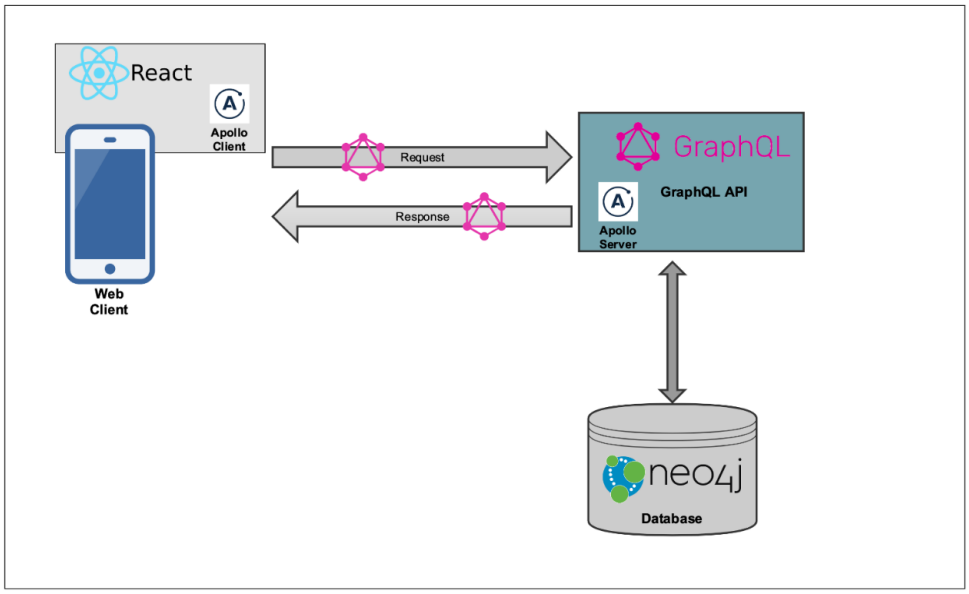


Figure 1.1 The components of GRANDstack include GraphQL, React, Apollo, and Neo4j database.

The focus of this book is learning how to build applications with GraphQL, so as we cover GraphQL it will be done in the context of building an application and using it with other technologies: how to design our schema, how to integrate with a database, how to build a web application that can query our GraphQL API, how to add authentication to our application, and so on. As a result, this book assumes basic knowledge of how web applications are typically built but doesn't necessarily require experience with each specific technology. To be successful, the reader should have a basic familiarity with JavaScript, both client-side and Node.js, and be familiar with concepts such as REST APIs and databases. You should be familiar with the `npm` command-line tool (or `yarn`) and how to use it to create Node.js projects and install dependencies. We include a brief introduction to each technology when it's introduced and suggest other resources for more in-depth coverage where needed by the reader. It's also important to note that we'll cover specific technologies to be used alongside GraphQL and that at each phase a similar technology could be substituted (for example, we could build our front-end using Vue instead of React). Ultimately, the goal of this book is to show how these technologies fit together and provide a full-stack framework for building applications with GraphQL.

1.1 GraphQL

At its heart, GraphQL is a specification for building APIs. The GraphQL specification describes an API query language and a way for fulfilling those requests. When building a GraphQL API, we describe the data available using a strict type system. These type definitions become the specification for the API, and the client is free to request the data it requires based on these type definitions, which also define the entry points for the API.

GraphQL is typically framed as an alternative to REST, which is the API paradigm you're mostly likely to be familiar with. This can be true in some cases, however GraphQL can also wrap existing REST APIs or other data sources. This is due to the benefit of GraphQL being data-layer agnostic—we can use GraphQL with any data source.

> *GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for exactly what they need and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.*
>
> — graphql.org

Let's dive into several of the specific aspects of GraphQL.

1.1.1 GraphQL type definitions

Rather than being organized around endpoints that map to resources (as with REST), GraphQL APIs are centered around type definitions that define the data types, fields, and how they're connected in the API. These type definitions become the schema of the API, which is served from a single endpoint.

Because GraphQL services can be implemented in any language, a language-agnostic Schema Definition Language (SDL) is used to define GraphQL types. Let's look at an example, motivated by considering a simple movies application. Imagine you're hired to create a website that allows users to search a movie catalog for movie details such as title, actors, and description, as well as show recommendations for similar movies the user may find interesting (figure 1.2).

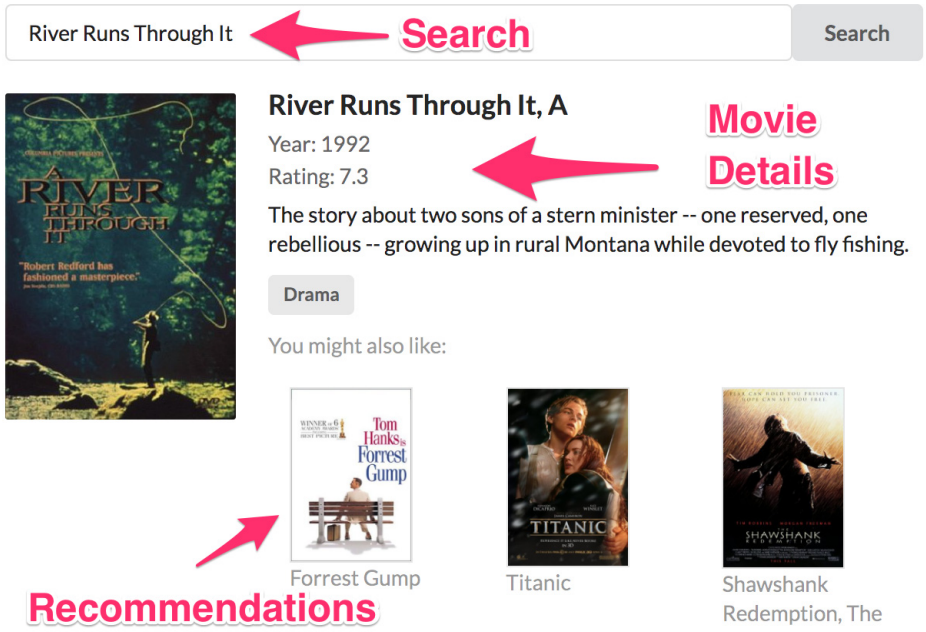


Figure 1.2 A simple GRANDstack movies application.

Let's start by creating simple GraphQL type definitions that define the data domain of our application, as shown in the following listing.

Listing 1.1 Simple GraphQL type definitions for a movies GraphQL API

```
type Movie {
  movieId: ID!
  title: String
  actors: [Actor]
}

type Actor {
  actorId: ID!
  name: String
  movies: [Movie]
}

type Query {
  allActors: [Actor]
  allMovies: [Movie]
  movieSearch(searchString: String!): [Movie]
  moviesByTitle(title: String!): [Movie]
}
```

`Movie` is a GraphQL object type, which means a type that contains one or more fields.

`Title` is a field on the `Movie` type.

Fields can reference other types, such as a list of `Actor` objects in this case.

`actorId` is a required (or non-nullable) field on the `Actor` type, which is indicated by the `!` character.

The `Query` type is a special type in GraphQL which indicate the entry points for the API.

Fields can also have arguments, in this case the `movieSearch` field takes a required String argument, `searchString`.

Our GraphQL type definitions declare the types used in the API, their fields, and how they're connected. When defining an object type (such as `Movie`) all fields available on the object and the type of each field is also specified (we can also add fields later, using the `extend` keyword). In this case we define `title` to be a scalar `String` type—a type that resolves to a single value, as opposed to an object type. Here `actors` is a field on the `Movie` type that resolves to an array of `Actor` objects, indicating that the actors and movies are connected (the foundation of the "graph" in GraphQL).

Fields can be either optional or required. The `actorId` field on the `Actor` object type is required (or non-nullable). Every `Actor` type must have a value for `actorId`. Fields that don't include a `!` are nullable, meaning values for those fields are optional.

The fields of the `Query` type become the entry points for queries into the GraphQL service. GraphQL schemas may also contain a `Mutation` type, which defines the entry points for write operations into the API. A third special entry point related type is the `Subscription` type, which defines events to which a client can subscribe.

> **NOTE** We're skipping over many important GraphQL concepts here, such as mutation operations, interface and union types, and so on, but don't worry, we're getting started and will get to these soon enough!

At this point you may wonder where the "graph" is in "GraphQL". It turns out we've defined a graph using our GraphQL type definitions earlier in this chapter. A graph is a data structure composed of nodes (the entities or objects in our data model) and relationships that connect nodes, which is exactly the structure we've defined in our

type definitions using SDL. The GraphQL type definitions above have defined a simple graph with this structure (figure 1.3).

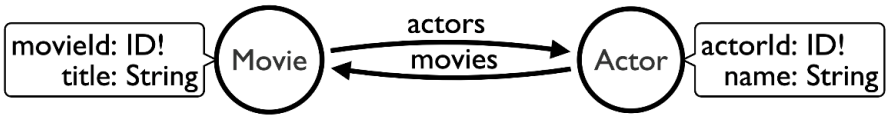


Figure 1.3 GraphQL type definitions, expressed as a graph.

Graphs are all about describing connected data and here we've defined how our movies and actors are connected in a graph.

When a GraphQL service receives a query, it's validated and executed against the GraphQL schema defined by these type definitions. Let's look at an example query that could be executed against a GraphQL service defined using the previous type definitions.

1.1.2 Querying with GraphQL

GraphQL queries define a traversal through the graph defined by our type definitions as well as requesting a subset of fields to be returned by the query; this is known as the selection set. In this query we start from the `allMovies` entry point and traverse the graph to find actors connected to each movie. Then for each of these actors, we traverse to all the other movies they're connected to. Our GraphQL query is shown in the following listing.

Listing 1.2 Simple GraphQL query

Optional naming of operation, `query` is the default operation and can therefore be omitted. Naming the query, in this case `FetchSomeMovies` is also optional and can be omitted.

```
query FetchSomeMovies {
  allMovies {
    title
    actors {
      name
      movies {
        title
      }
    }
  }
}
```

Here we specify the entry point, which is a field on either the `Query` or `Mutation` type, in this case our entry point for the query is the `allMovies` query field.

The selection set defines the fields to be returned by the query.

In the case of object fields, a nested selection set is used to specify the fields to be returned.

A further nested selection set is needed for the fields on `movies` to be returned.

Note that our query is nested and describes how to traverse the graph of related objects (in this case movies and actors), as in figure 1.4 and listing 1.3.

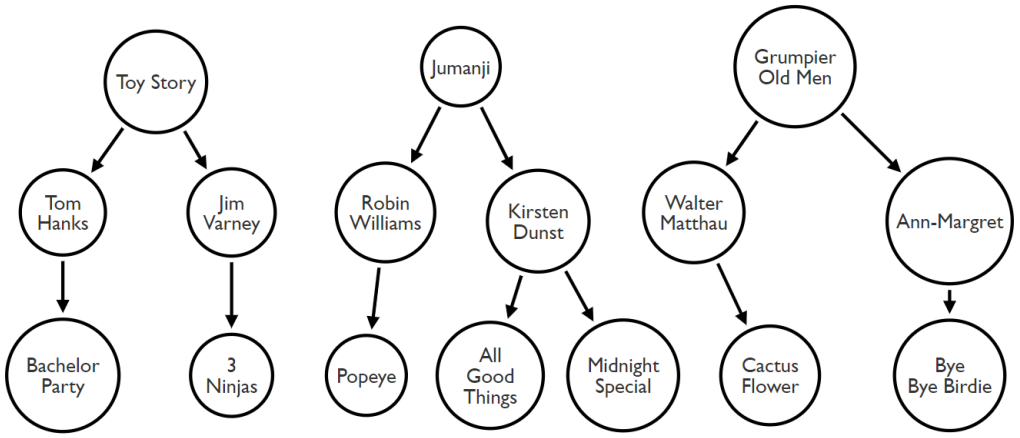


Figure 1.4 A GraphQL query traversal through the data graph.

Listing 1.3 Query results

```
"data": {
  "allMovies": [
    {
      "title": "Toy Story",
      "actors": [
        {
          "name": "Tom Hanks",
          "movies": [
            {
              "title": "Bachelor Party"
            }
          ]
        },
        {
          "name": " Jim Varney",
          "movies": [
            {
              "title": "3 Ninjas: High Noon On Mega Mountain"
            }
          ]
        }
      ]
    },
    {
      "title": "Jumanji",
      "actors": [
        {
          "name": "Robin Williams",
          "movies": [
            {
              "title": "Popeye"
            }
          ]
```

```
        },
        {
          "name": "Kirsten Dunst",
            "movies": [
              {
                "title": "Midnight Special"
              },
              {
                "title": "All Good Things"
              }
            ]
        }
      ]
    },
    {
      "title": "Grumpier Old Men",
      "actors": [
        {
          "name": "Walter Matthau",
          "movies": [
            {
              "title": "Cactus Flower"
            }
          ]
        },
        {
          "name": " Ann-Margret",
          "movies": [
            {
              "title": "Bye Bye Birdie"
            }
          ]
        }
      ]
    }
  ]
}
```

As you can see from the results, the response matches the form of the query—exactly the fields requested in the query are returned.

But where does the data come from? The data fetching logic for GraphQL APIs is defined in functions called resolvers, which contain the logic for resolving the data for an arbitrary GraphQL request from the data layer. GraphQL is data-layer agnostic so the resolvers could query one or more databases or even fetch data from another API, even a REST API. We cover resolvers in depth in the next chapter.

1.1.3 Advantages of GraphQL

Now that we've seen our first GraphQL query you may wonder, "OK that's nice, but I can fetch data about movies with REST, too. What's so great about GraphQL?". Let's review some of the benefits of GraphQL

OVERFETCHING AND UNDERFETCHING

Overfetching refers to a pattern commonly associated with REST where unnecessary and unused data is sent over the network in response to an API request. Because REST is modeling resources, when we make a `GET` request for say `/movie/tt0105265` we get back the representation of that movie, nothing more nothing less, as shown in the following listing.

Listing 1.4 REST API response for GET /movie/tt0105265

```
{
  "title": "A River Runs Through It",
  "year": 1992,
  "rated": "PG",
  "runtime": "123 min",
  "plot": "The story about two sons of a stern minister -- one reserved, one
     rebellious -- growing up in rural Montana while devoted to fly
     fishing.",
  "movieId": "tt0105265",
  "actors": ["nm0001729", "nm0000093", "nm0000643", "nm0000950"],
  "language": "English",
  "country": "USA",
  "production": "Sony Pictures Home Entertainment",
  "directors": ["nm0000602"],
  "writers": ["nm0533805", "nm0295030"],
  "genre": "Drama",
  "averageReviews": 7.3
}
```

But what if the view of our application only needs to render the title and year of the movie? Then we've unnecessarily sent too much data over the wire. Further, several of those movie fields may be expensive to compute. For example, if we need to calculate `averageReviews` by aggregating across all movie reviews for each request, but we're not even showing that in the application view, that's wasted compute time and unnecessarily impacts the performance of our API (in the real world we may cache this, but this is an example).

Similarly, underfetching is a pattern associated with REST where not enough data is returned by the request.

Let's say our application view needs to render the name of each actor in a movie. First, we make a GET request for `/movie/tt0105265`. As seen previously, we have an array of IDs for the actors connected to this movie. Now to get the data required for our application we need to iterate over this array of actor IDs to get the name of each actor to render in our view:

```
/actor/nm0001729
/actor/nm0000093
/actor/nm0000643
/actor/nm0000950
```

With GraphQL we can accomplish this in a single request, solving both the overfetching and underfetching problems. This results in improved performance on the server

side because we're spending less compute resources at the data layer, less overall data sent over the wire, and reduced latency by being able to render our application view with a single network request to the API service.

GraphQL specification

GraphQL is a specification for client-server communication that describes the features, functionality, and capability of the GraphQL API query language. Having this specification gives a clear guide of how to implement your GraphQL API and clearly defines what is and what isn't GraphQL.

REST doesn't have a specification, instead there are many different implementations, from REST-ish to Hypermedia As The Engine Of Application State (HATEOAS). Having a specification as part of GraphQL simplifies debates over endpoints, status codes, and documentation. All this comes built in with GraphQL which leads to productivity wins for developers an API designer. The specification provides a clear path for API implementors.

With GraphQL it's graphs all the way down

REST models itself as a hierarchy of resources, yet most interactions with APIs are done in terms of relationships. For example, given our movie query, for this movie, show me all of the actors connected to it, and for each actor show me all the other movies they've acted in—we're querying for relationships.

GraphQL can also help unify data from disparate systems. Because GraphQL is data-layer agnostic we can build GraphQL APIs that integrate data from multiple services or microservices together and provide a clear way to integrate data from these different system into a single GraphQL schema.

GraphQL can also be used to compartmentalize data fetching in the application in a component-based data interaction pattern. Because each GraphQL query can describe exactly the graph traversal and fields to be returned, encapsulating these queries with application components can help simplify application development and testing. We'll see how to apply this once we start building our React application.

Introspection

Introspection is a powerful feature of GraphQL that allows us to ask a GraphQL API for the types and queries it supports. Introspection becomes a way of self-documenting the API. Tools that make use of introspection can provide human readable API documentation, visualization tooling, and leverage code generation to create API clients.

1.1.4 Disadvantages of GraphQL

GraphQL isn't a silver bullet, and we shouldn't think of GraphQL as the solution to all of our API-related problems. One of the most notable challenges of adopting GraphQL is that several well-understood practices from REST don't apply when using GraphQL. For example, HTTP status codes are commonly used to convey success, failure, and other cases of a REST request: 200 OK means our request was successful and 404 Not Authorized means we forgot an auth token or don't have the correct permis-

sions for the resource requested. However, with GraphQL, each request returns 200 OK regardless if the request was a complete success or not. This makes error handling a bit different in the GraphQL world. Instead of a single status code describing the result of our request, GraphQL errors are typically returned at the field level of the selection set. This means that we may have successfully retrieved a part of our GraphQL query, while other fields returned errors.

Web caching is another well understood area from REST that's handled a bit differently with GraphQL. With REST, caching the response for `/movie/123` is possible because we can return the same exact result for each GET request. This isn't possible with GraphQL because each request could contain a different selection set, meaning we can't simply return a cached result. This is mitigated by most GraphQL clients implementing client caches at the application level and in practice most of the time our GraphQL requests are in an authenticated environment where caching isn't applicable.

Another challenge is that of exposing arbitrary complexity to the client and related performance considerations. If the client is free to compose queries as they wish, how can we ensure these queries don't become so complex as to impact performance significantly or overwhelm the computing resources of our backend infrastructure? Fortunately, GraphQL tooling allows us to restrict the depth of the queries used and further restrict the queries that can be run to a whitelisted selection of queries (known as persisted queries). A related challenge is implementing rate limiting. With REST we could restrict the number of requests that can be made in a certain time period; however, with GraphQL this becomes more complicated since the client could be requesting multiple objects in a single query. This results in bespoke query costing implementations to address rate limiting.

Finally, the so called "n+1 query problem" is a common problem in GraphQL data fetching implementations that can result in multiple round trips to the database and negatively impact performance. Consider the case where we request information about a movie and all actors of the movie. In the database we might store a list of actor IDs associated with each movie, which is returned with our request for the movie details. In a naive GraphQL implementation we then need to retrieve the actor details, and we must make a separate request to the database for each actor object, resulting in a total of n (the number of actors) + 1 (the movie) queries to the database. To address the n+1 query problem, tools such as DataLoader allow us to batch requests to the database, increasing performance, and database integrations such as neo4j-graphql.js and PostGraphile allow us to generate a single database query from an arbitrary GraphQL request.

GraphQL limitations

It's important to understand that GraphQL is an API query language and not a database query language. GraphQL lacks semantics for many complex operations required of database query languages, such as aggregations, projects, and variable length path traversals.

1.1.5 GraphQL tooling

In this section we review GraphQL specific tooling that will help us build, test, and query our GraphQL API.

GRAPHIQL

GraphiQL is an in-browser tool for exploring and querying GraphQL APIs. With GraphQL we can execute GraphQL queries against a GraphQL API and view the results. Thanks to GraphQL's introspection feature we can view the types, fields, and queries supported by the GraphQL API we've connected to. In addition, because of the GraphQL type system we have immediate query validation as we construct our query (figure 1.5).

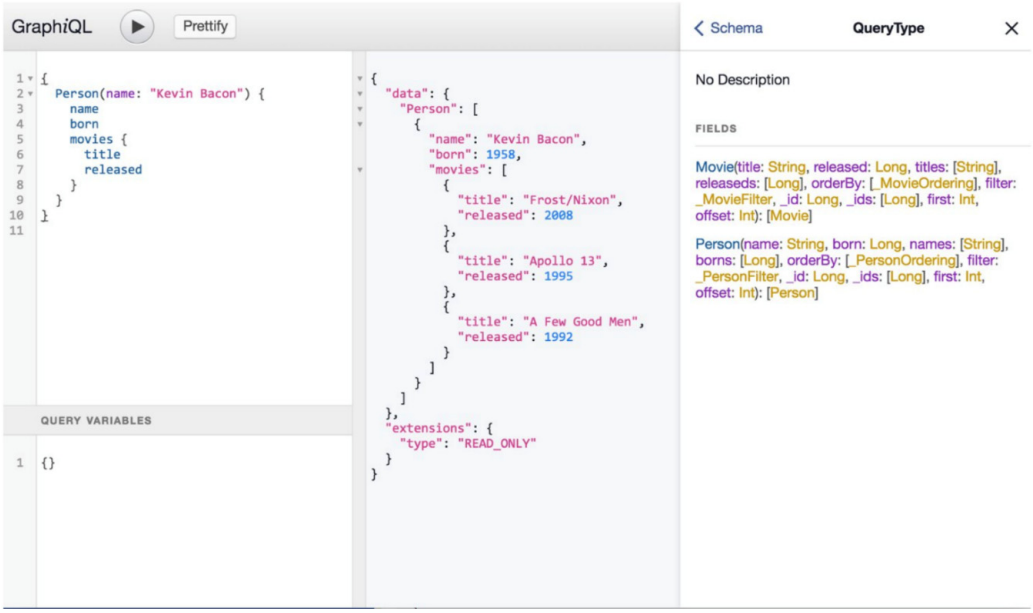


Figure 1.5 GraphiQL screenshot.

GRAPHQL PLAYGROUND

Similar to GraphiQL, GraphQL Playground (figure 1.6) is an in-browser tool for executing GraphQL queries, viewing the results, and exploring the GraphQL API's schema, powered by GraphQL's introspection features. GraphQL Playground has a few additional features such as viewing GraphQL type definitions, searching through the GraphQL schema, and easily adding request headers (such as those required for authentication). We include GraphQL Playground in addition to GraphiQL because certain tools (such as Apollo Server) expose GraphQL Playground by default, while others expose GraphiQL.

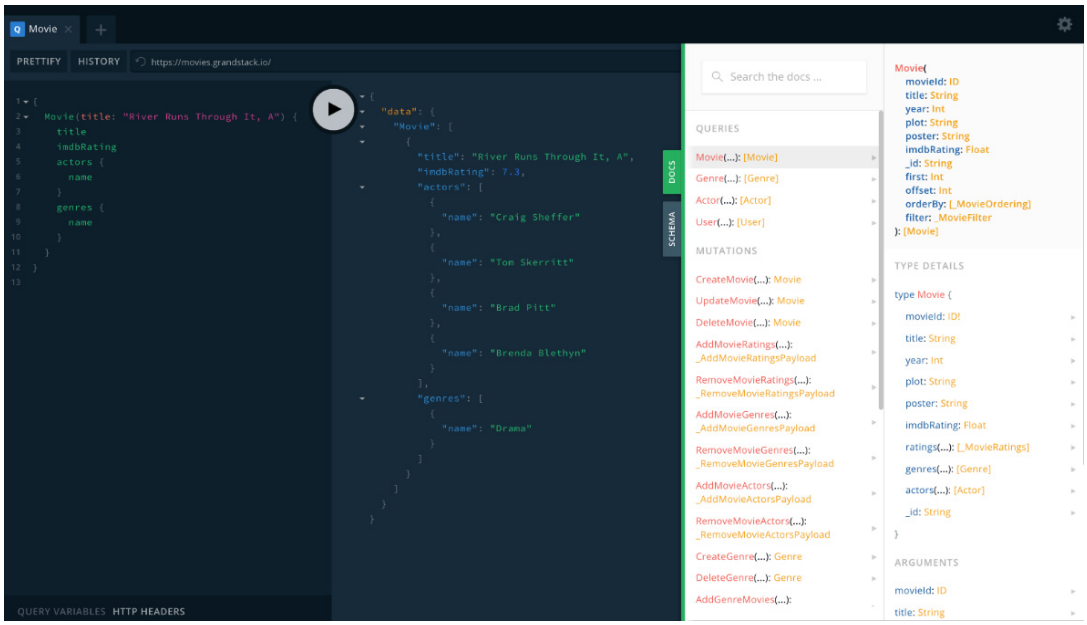


Figure 1.6 GraphQL Playground screenshot.

1.2 React

React is a declarative, component-based library for building user interfaces using JavaScript. React uses a virtual DOM (a copy of the actual Document Object Model) to efficiently calculate DOM updates required to render views as state and data changes. This means users design views that map to application data and React handles rendering the DOM updates efficiently. Components encapsulate data handling and user interface rendering logic without exposing their internal structure so they can be easily composed together to build complex applications.

1.2.1 React components

Let's examine a simple React component in the following listing.

Listing 1.5 A simple GraphQL component

Components can be either class components or functional components. Here we implement a class component, inheriting from the `React.Component` class.

```
class MovieTitleComponent extends React.Component {
  constructor() {
    super()
    this.state = {title: 'River Runs Through It, A'}
  }

  render() {
    return (
```

The constructor for our class component is called when our component is first initialized.

Class components can store data in state. We can set the initial state value in the constructor.

The `render` method defines what user interface elements the component will render.

```
      <div>{this.state.title}</div>
    )
  }
}
```

Here we access the `title` value from our component state and render that inside a `div` tag.

COMPONENT LIBRARIES

Because components encapsulate data handling and user interface rendering logic and are easily composable, it's practical to distribute libraries of components that can be used as dependencies of your project for quickly leveraging complex styling and user interface design. We'll use the Material-UI component library that will allow us to import many popular common user interface components such as a grid layout, data table, navigation, and inputs.

1.2.2 JSX

React is typically used with a JavaScript language extension called JSX. JSX looks similar to XML and is a powerful way of building user interfaces in React and composing React components. It is possible to use React without JSX, but most users prefer the readability and maintainability that JSX offers. We'll introduce JSX in chapter 5.

We'll cover React concepts such as unidirectional data flow, props and state, and data fetching in chapter 5.

1.2.3 React tooling

Here we review useful tooling that will help us build, develop, and troubleshoot React applications.

CREATE REACT APP

Create React App is a command line tool that can be used to quickly create the scaffolding for a React application, taking care of configuring build settings, installing dependencies, and templating a simple React application to get started.

REACT CHROME DEVTOOLS

React Chrome Devtools is a browser integration that lets us inspect a React application, seeing under the hood the component hierarchy and the props and state of each component while our application is running (figure 1.7).

1.3 Apollo

Apollo is a collection of tooling to make it easier to use GraphQL, both on the server and the client. We'll use Apollo Server, a Node.js library for building our GraphQL API and Apollo Client, a client-side JavaScript library for querying our GraphQL API from our application.

1.3.1 Apollo Server

Apollo Server allows us to easily spin up a Node.js server serving a GraphQL endpoint by defining our type definitions and resolver functions. Apollo Server can be used with many different web frameworks; however, the default and most popular is

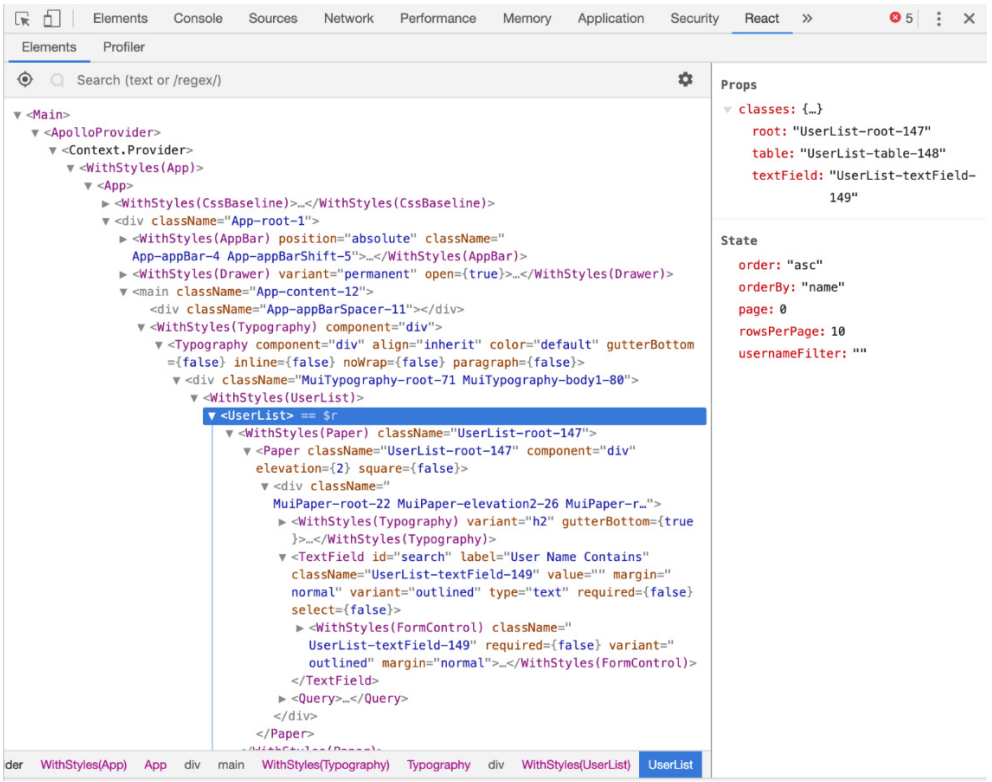


Figure 1.7 **React Chrome Devtool.**

Express.js. Apollo Server can also be used with serverless functions such as Amazon Lambda and Google Cloud Functions.

Apollo Server can be installed with `npm`: `npm install apollo-server`.

1.3.2 Apollo Client

Apollo Client is a JavaScript library for querying GraphQL APIs and has integrations with many frontend frameworks, including React, Vue.js as well as native mobile versions for iOS and Android. We'll use the React Apollo Client integration to implement data fetching via GraphQL in our React components. Apollo Client handles client data caching and can also be used to manage local state data.

The React Apollo Client library can be installed with `npm`: `npm install react-apollo`.

1.4 Neo4j database

Neo4j is an open-source native graph database. Unlike other databases that use tables or documents for the data model, the data model used with Neo4j is a graph, specifically known as the Property Graph data model, and allows us to model, store, and

query our data as a graph. Graph databases such as Neo4j are optimized for working with graph data and executing complex graph traversals, such as those defined by GraphQL queries.

One of the benefits of using a graph database with GraphQL is that we maintain the same data model throughout our application stack, working with graphs on both the frontend, API, and backend. Another benefit has to do with the performance optimizations graph databases make versus other database systems, such as relational databases. Many GraphQL queries end up nested several levels deep—the equivalent of a `JOIN` operation in a relational database. Graph databases are optimized for performing these graph traversal operations efficiently, so a graph database is a natural fit for the backend of a GraphQL API.

> **NOTE** It's important to note that we aren't querying the database directly with GraphQL. While there are database integrations for GraphQL, it's important to realize the GraphQL API is a layer that sits between our application and the database.

1.4.1 Property graph data model

Like many graph databases Neo4j uses a property graph model. The components of the property graph model are (figure 1.8):

- Nodes: The entities, or objects, in our data model
- Relationships: Connections between nodes
- Labels: A grouping semantic for nodes
- Properties: Key-value pair attributes, stored on nodes and relationships

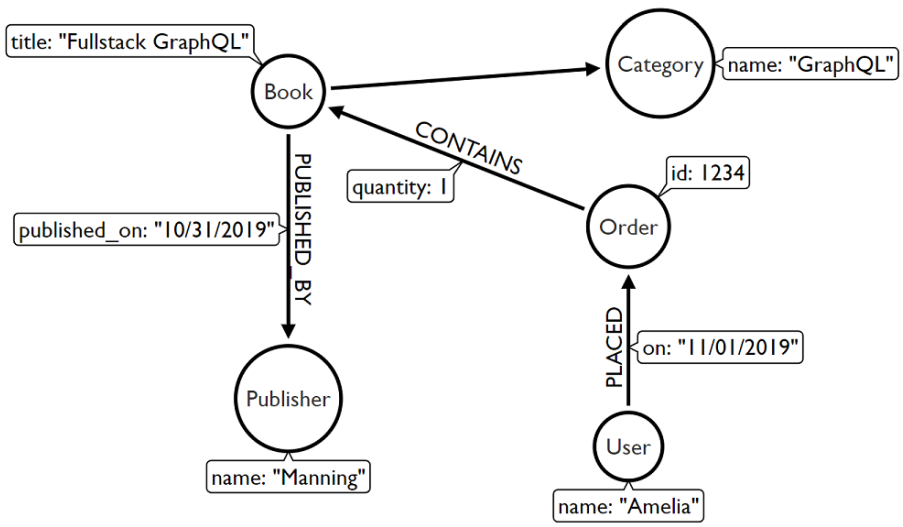


Figure 1.8 Property graph example modeling an order for a book placed by a user and how the data is connected as a graph.

The property graph data model allows us to express complex, connected data in a flexible way. This data model also has the additional benefit of closely mapping to the way we often think about data when dealing with a domain.

1.4.2 Cypher query language

Cypher is a declarative graph query language, used by Neo4j and other graph databases and graph compute engines. You can think of Cypher as similar to SQL, but instead designed for graph data. A major feature of Cypher is that of pattern matching. With graph pattern matching in Cypher, we can define the graph pattern using ASCII-art like notation. Cypher is an open standard through the openCypher project. Let's look at a simple Cypher example in the following listing, querying for movies and actors connected to these movies.

Listing 1.6 Simple Cypher query

`MATCH` is used to search for a graph pattern described using an ASCII-art like notation. In the pattern, nodes are defined within parentheses, for example `(m:Movie)`. The `:Movie` indicates we should match nodes with the label `Movie` and the `m` before the colon becomes a variable that is bound to any nodes that match the pattern. We can refer to `m` later throughout the query. Relationships are defined by square brackets, for example `-[r:ACTED_IN]-` and follow a similar convention where `:ACTED_IN` declares the `ACTED_IN` relationship type and `r` becomes a variable we can refer to later in the query to represent any relationships matching that pattern.

```
MATCH (m:Movie)<-[r:ACTED_IN]-(a:Actor)
RETURN m,r,a #B
```

In the `RETURN` clause we specify the data to be returned by the query. Here we specify the variables `m`, `r`, and `a`, variables that were defined in the `MATCH` clause above.

Cypher is an open source query language through the openCypher project and several other graph databases and graph systems implement Cypher.

1.4.3 Neo4j tooling

We'll use Neo4j Desktop for managing our Neo4j instances locally, and Neo4j Browser, a developer tool for querying and interacting with our Neo4j database. For querying Neo4j from our GraphQL API we will use the JavaScript Neo4j client driver as well as neo4j-graphql.js, a GraphQL integration for Neo4j.

NEO4J DESKTOP

Neo4j Desktop is Neo4j's command center. From Neo4j Desktop (figure 1.9) we can manage Neo4j database instances, including editing configuration, installing plugins and graph apps (such as visualization tools), and access admin level features such as dump/load database. Neo4j Desktop is the default download experience for Neo4j.

NEO4J BROWSER

Neo4j Browser (figure 1.10) is a query workbench for Neo4j. With Neo4j Browser we can query the database with Cypher and visualize the results, either as a graph visualization or tabular results.

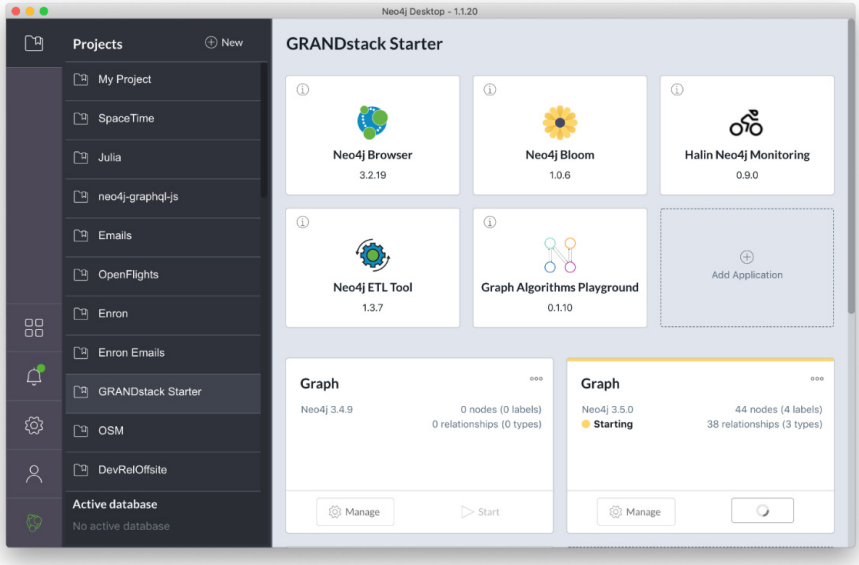


Figure 1.9 Neo4j Desktop.

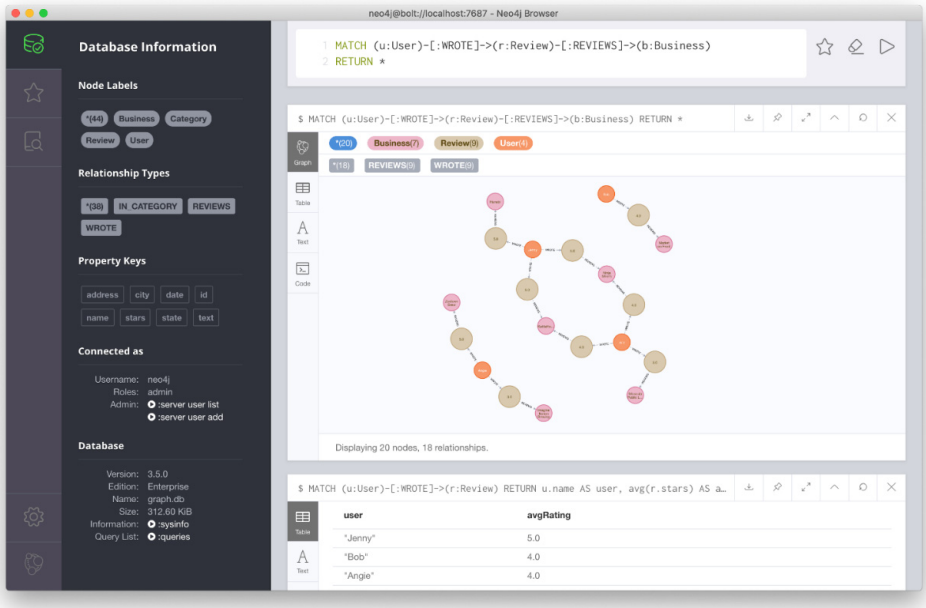


Figure 1.10 Neo4j Browser.

NEO4J CLIENT DRIVERS

Because our end goal is to build an application that talks to our Neo4j database, we'll use the language drivers for Neo4j. Client drivers are available in many languages (Java, Python, .Net, JavaScript, etc.) but we'll use the Neo4j JavaScript driver (listing 1.7).

> **NOTE** The Neo4j JavaScript driver has both a node.js and browser version (allowing connections to the database directly from the browser); however, in this book we only use the node.js version.

```
npm install neo4j-driver
```

Listing 1.7 Basic Neo4j JavaScript driver usage

```
const neo4j = require("neo4j-driver");

const driver = neo4j.driver("bolt://localhost:7687",
  neo4j.auth.basic("neo4j", "letmein"),
  {encrypted: true});

const session = driver.session();

session.run("MATCH (n) RETURN COUNT(n) AS num")
.then(result => {
  const record = result.records[0];
  console.log(`Your database has ${record['num']} nodes`);
})
.catch(error => {
  console.log(error);
})
.finally( () => {
  session.close();
)
```

Importing the module.

Creating a driver instance, specifying the database connection string, using the bolt protocol.

Specifying the database user and password.

Driver configuration options.

Sessions are more lightweight and should be instantiated for a given block of work.

Run the query in an auto-commit transaction, returns a Promise.

The promise resolver to a resultset.

Accessing the records of the resultset.

Be sure to close the session.

We'll learn how to use the Neo4j JavaScript client driver in our GraphQL resolver functions to implement data fetching in our GraphQL API.

NEO4J-GRAPHQL.JS

The neo4j-graphql.js library is a GraphQL to Cypher query execution layer for Neo4j. It works with any of the JavaScript GraphQL server implementations such as Apollo Server. We'll learn how to use this library to:

1. Use GraphQL type definitions to drive the Neo4j database schema.
2. Generate a full CRUD GraphQL API from GraphQL type definitions.
3. Generate a single Cypher database query for arbitrary GraphQL requests.
4. Implement custom logic defined in Cypher.

While GraphQL is data layer agnostic—you can implement a GraphQL API using any data source or database—when used with a graph database, you have benefits such as

Other implementations of Neo4j-GraphQL

Similar features are available for the Java/JVM ecosystem with neo4j-graphql-java—a library that can be used with Java- and JVM-based GraphQL servers—or language-agnostic with the Neo4j GraphQL database plugin, an extension for neo4j that makes available a GraphQL endpoint served by the database, as well as procedures for GraphQL schema management.

reducing mapping and translation of the data model and performance optimizations for addressing complex traversals defined with GraphQL.

1.5 How it all fits together

Now that we've taken a look at each individual piece of GRANDstack, let's see how everything fits together in the context of a full-stack application. Throughout this chapter, as we've looked at examples of each technology, we've used simple movies data related examples. Let's see how a simple GRANDstack movies search application would work behind the scenes.

Our application has three simple requirements:

1. Allow the user to search for a movie by title
2. Display to the user any matching results and details (rating, genres, etc.) of those movies
3. Show a list of similar movies that might be a good recommendation if the user liked the matching movie

If this application were built using GRANDstack, here's how the different components would fit together following the flow of a request from the client application, searching for movies by title, to the GraphQL API, then resolving data from the Neo4j database, and back to the client, rendering the results in an updated user interface view (figure 1.11).

1.5.1 React and Apollo Client—making the request

The frontend of our application is built in React; specifically we have a `MovieSearch` React component that renders a text box that accepts user input (a movie search string to be provided by the user). This `MovieSearch` component also contains the logic for taking the user input, combining it with a GraphQL query, and sending this query to the GraphQL server to resolve the data using the Apollo Client React integration.

The following listing shows what the GraphQL query sent to the API might look like if the user searched for "River Runs Through It".

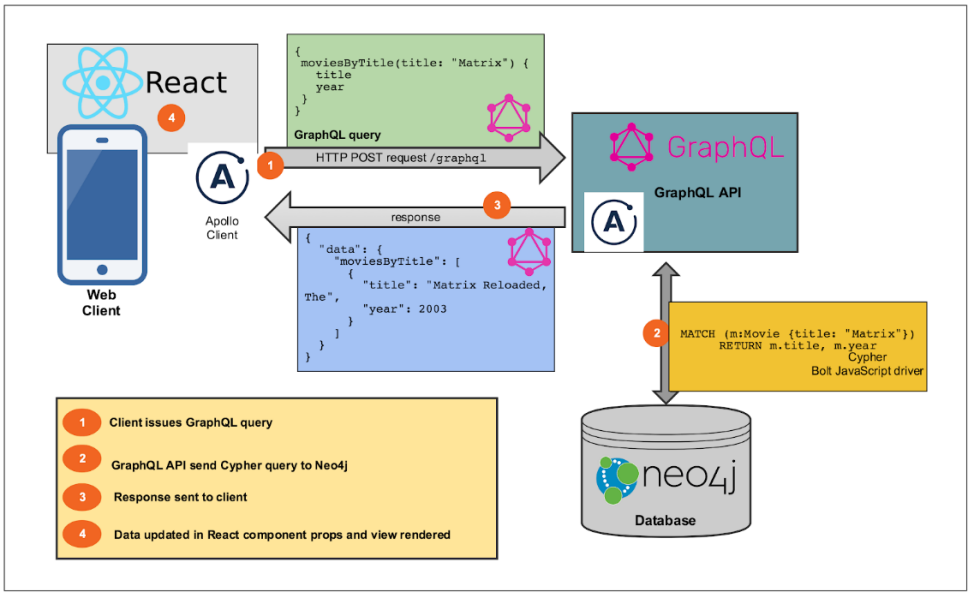


Figure 1.11 Following a movie search request through a GRANDstack application.

Listing 1.8 GraphQL query searching for movies matching "River Runs Through It"

```
{
  moviesByTitle(title: "River Runs Through It") {
    title
    poster
    imdbRating
    genres {
      name
    }
    recommendedMovies {
      title
      poster
    }
  }
}
```

This data fetching logic is enabled by Apollo Client, which we use in the `MovieSearch` component. Apollo Client implements a cache, so when the user enters their search query, Apollo Client first checks the cache to see if a GraphQL query has previously been handled for this search string. If not, then the query is sent to the GraphQL server as an HTTP POST request to `/graphql`.

1.5.2 Apollo Server and GraphQL backend

The backend for our movies application is a Node.js application that uses Apollo Server and the Express web server library to serve a `/graphql` endpoint over HTTP. An HTTP GraphQL server is composed of a network layer and a GraphQL schema layer. The net-

work layer is responsible for processing HTTP requests, extracting the GraphQL operation, and returning HTTP responses. The GraphQL schema layer includes the GraphQL type definitions, which define the entry points and data structures for the API, as well as resolver functions which are responsible for resolving the data from the data layer. The data layer could be one or more databases, another service, other APIs or any combination of these. By default, Apollo Server uses Express as the network layer.

When Apollo Client makes its request our GraphQL server handles the request by validating the query and then begins to resolve the request by first calling the root level resolver function, which in this case is `moviesByTitle`. This resolver function is passed the `title` argument—the value the user typed into the search text box. Inside our resolver function we have the logic for querying the database to find movies whose titles match the search query, retrieving the movie details, and for each matching movie finding a list of other recommended movies.

Resolver implementation

Throughout this book we show three methods for implementing resolver functions:

1. The "naive" approach of defining database queries inside resolvers
2. Using the Apollo DataSource library to collocate data fetching code, which has the benefit of implementing server-side caching
3. Auto-generating resolvers using GraphQL "engine" libraries such as neo4j-graphql.js.

This example covers only the first case.

Resolver functions (figure 1.12) are executed in a nested fashion, in this case starting with the `moviesByTitle` Query field resolver, which is the root level resolver for this operation. The `moviesByTitle` resolver will return a list of movies, then the resolver for each field requested in the query will be called and passed an item from the list of movies returned by `moviesByTitle`—essentially iterating over this list of movies.

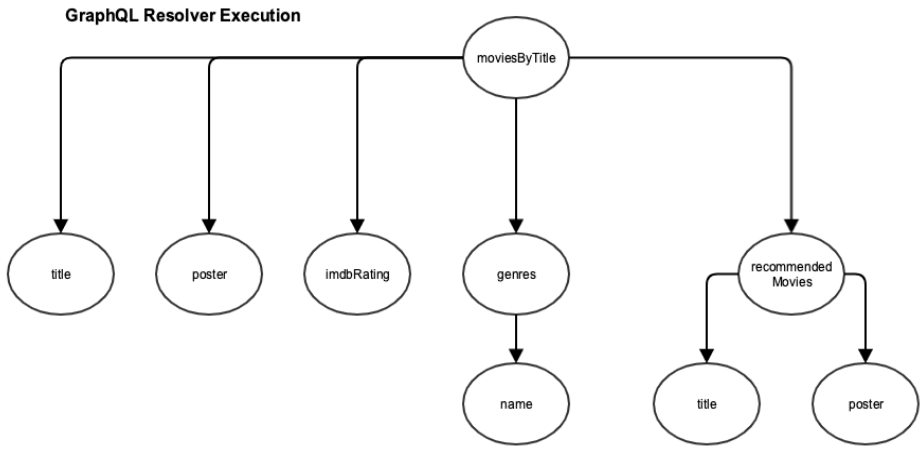


Figure 1.12 GraphQL resolver functions are called in a nested fashion.

Each resolver function contains logic for resolving data for a piece of the overall GraphQL schema. For example, the `recommendedMovies` resolver when given a movie has the logic to find similar movies that the viewer might also enjoy. In this case, this is done by querying the database using a simple Cypher query to search for users that have viewed the movie and traverses out to find other movies those users have viewed, a simple collaborative filtering recommendation. This query is executed in Neo4j using the Node.js JavaScript Neo4j client driver, as shown in the following listing.

Listing 1.9 A simple movie recommendation query

```
MATCH (m:Movie {movieId: $movieID})<-[:RATED]-(:User)-[:RATED]->(rec:Movie)
WITH rec, COUNT(*) AS score ORDER BY score DESC
RETURN rec LIMIT 3
```

n+1 query problem

Here we have a perfect demonstration of the "n+1 query problem". Our root-level resolver is returning a list of movies. Now, to resolve our GraphQL query we need to call the `actors` resolver, once for each movie. This results in multiple requests to the database, which can impact performance.

Ideally, we instead make a single request to the database, which contains fetches all data needed to resolve the GraphQL query in a single request. Here are a few solutions to this problem:

1. The DataLoader library allows us to batch our requests together.
2. GraphQL "engine" libraries, such as neo4j-graphql.js can generate a single database query from an arbitrary GraphQL request, leveraging the advantage of the "graph" nature of GraphQL without negative performance impacts from multiple database calls.

1.5.3 React and Apollo Client: handling the response

Once our data fetching is complete the data is sent back to Apollo Client, the cache is updated so if this same search query is executed in the future, the data will be retrieved from the cache, instead of requesting the data from the GraphQL server.

Our `MovieSearch` React component passes the results of the GraphQL query to a `MovieList` component as props, which in turn renders a series of `Movie` components, updating the view to show the movie details for each matching movie, in this case, one. And our user is presented with a list of movie search results (figure 1.13)!

The goal of the previous example is to show how GraphQL, React, Apollo, and Neo4j Database are used together to build a simple full-stack application. We've omitted many details, such as authentication, authorization, optimizing performance, but don't worry, we'll cover all this in detail throughout the book.

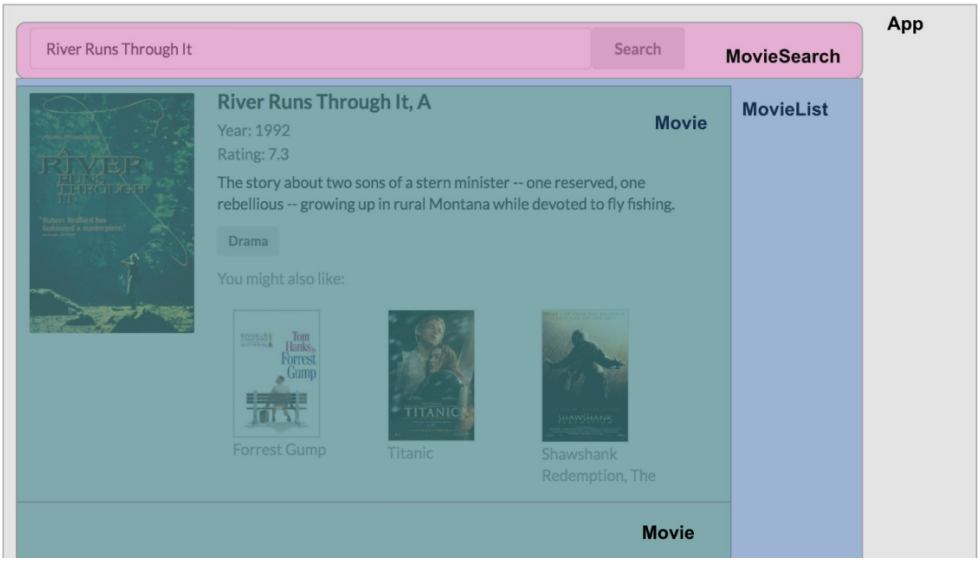


Figure 1.13 **React components are composed together to build a complex user interface.**

1.6 What we will build

The simple movie search example that we've used throughout the chapter was hopefully a decent introduction to the concepts we'll learn throughout the book. Instead of building a movie search application, let's start from scratch and build a new application, working through the requirements and GraphQL API design together as we build our knowledge of GraphQL. To demonstrate the concepts covered in this book, we'll build a web application that uses GRANDstack. This web application will be a simple business reviews application. The requirements of the application are:

- List businesses and business details
- Allow users to write reviews of businesses
- Allow users to search for businesses and show personalized recommendations to the user

To implement this application, we need to design our API, user interface, and database considerations, including authentication and authorization.

Exercises

1. To familiarize yourself with GraphQL and writing GraphQL queries, explore the public movies GraphQL API at https://movies.grandstack.io. Open the URL in a web browser to access GraphQL Playground and explore the `DOCS` and `SCHEMA` tab to see the type definitions.

   Try writing queries to answer the following questions:

   - Find the titles of the first 10 movies, ordered by title.
   - Who acted in the movie "Jurassic Park"?

   - What are the genres of "Jurassic Park"? What other movies are in those genres?
   - What movie has the highest `imdbRating`?
2. Consider the business reviews application we described earlier in the chapter. See if you can create the GraphQL type definitions necessary for this application.
3. Download Neo4j and familiarize yourself with Neo4j Desktop and Neo4j Browser. Work through a https://neo4jsandbox.com example dataset guide.

You can find solutions to the exercises, as well as code samples in the GitHub repository for this book: https://github.com/johnymontana/fullstack-graphql-book

Summary

- GRANDstack is a collection of technologies for building fullstack web applications with GraphQL and is composed of **G**raphQL, **R**eact, **A**pollo, and **N**eo4j **D**atabase.
- GraphQL is an API query language and runtime for fulfilling requests. We can use GraphQL with any data layer. To build a GraphQL API we first define the types, which includes the fields available on each type and how they are connected, describing the data graph.
- React is a JavaScript library for building user interfaces. We use JSX to construct components which encapsulate data and logic. These components can be composed together, allowing for building complex user interfaces.
- Apollo is a collection of tools for working with GraphQL, both on the client and the server. Apollo Server is a node.js library for building GraphQL APIs. Apollo Client is a JavaScript GraphQL client that has integrations for many frontend frameworks, including React.
- Neo4j is an open-source graph database that uses the Property Graph data model, which consists of nodes, relationships, labels, and properties. We use the Cypher query language for interacting with Neo4j.

3 Graphs in the database

This chapter covers

- Introducing graph databases with a focus on Neo4j
- Understanding the property graph data model
- Using the Cypher query language to create and query data in Neo4j
- Using client drivers for Neo4j, specifically the JavaScript Node.js driver

Fundamentally, a graph database is a software tool that allows the user to model, store, and query data as a graph. Working with a graph at the database level is often more intuitive for modeling complex connected data and can be more performant when working with complex queries that require traversing many connected entities.

In this chapter we begin the process of creating a property graph data model using the business requirements from the previous chapter and show how to model that in Neo4j. Then we compare that to the GraphQL model created in the previous chapter. We then explore the Cypher query language, focusing on how to write Cypher queries to address the requirements of our application. Along the way, we show how to install Neo4j, use Neo4j Desktop to create new Neo4j projects locally, and use Neo4j Browser to query Neo4j, and visualize the results. Finally, we show how to use the Neo4j JavaScript client driver to create a simple Node.js application that queries Neo4j.

3.1 Neo4j overview

Neo4j is a native graph database that uses the property graph model for modeling data and the Cypher query language for interacting with the database. Neo4j is a transactional database with full ACID guarantees and can also be used for graph analytics. Graph databases such as Neo4j are optimized for working with highly connected data and queries that traverse the graph (think of the equivalent of multiple JOINs in a relational database) and are therefore the perfect backend for GraphQL APIs that describe connected data and often result in complex, nested queries. Neo4j is open-source and can be downloaded from https://neo4j.com/download.

We'll use Neo4j Desktop and Neo4j Browser in this chapter as we learn how to create and query data in Neo4j, but first let's dig into the property graph model used by Neo4j and see how it relates to the model used to describe GraphQL APIs we reviewed in the previous chapter.

3.2 Graph data modeling with Neo4j

Unlike other databases that use tables or documents to model data, graph databases such as Neo4j model, store, and allow the user to query data as a graph. In a graph, nodes are the entities and relationships connect them (see figure 3.1). In a relational database, we represent relationships with foreign keys and join tables. In a document database, we reference other entities using ids or even denormalizing and embedding other entities in a single document.

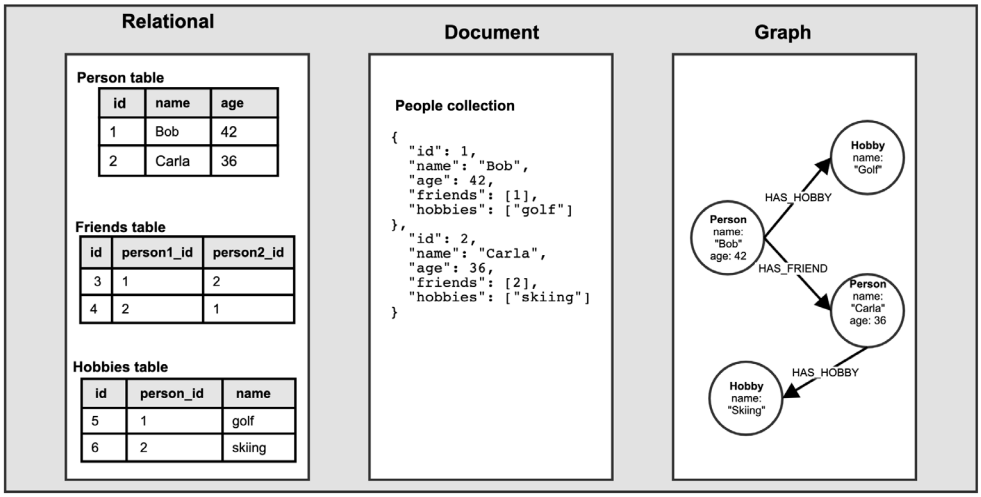


Figure 3.1 Comparing datamodels across relational, document, and graph.

The first step when working with a database is to determine the datamodel that will be used. In our case our datamodel will be driven from the business requirements we've defined in the previous chapter, working with businesses, users, and reviews. Review

the requirements listed in the first section of the previous chapter for a refresher. Let's take those requirements and our knowledge of the domain to create a "whiteboard model" (figure 3.2).

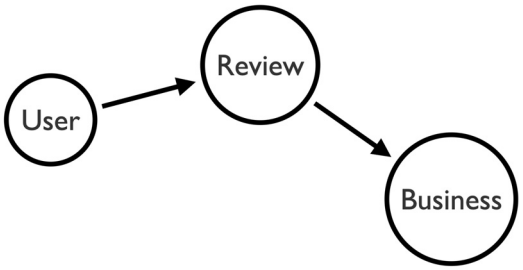


Figure 3.2 Building the Property Graph model: whiteboard model.

> **Whiteboard model**
> We'll use the term "whiteboard model" to refer to the diagram typically created when first reasoning about a domain, which is often a graph of entities and how they relate, drawn on a whiteboard.

How do we translate this mental model from the "whiteboard" model to the physical data model used by the database? In other systems, this might involve creating an ER diagram or defining the schema of the database. Neo4j is said to be "schema optional". While we can create database constraints to enforce data integrity, such as property uniqueness, we can also use Neo4j without these constraints or a schema. But the first step is to define a model using the Property Graph data model, which is the model used by Neo4j and other graph databases. Let's convert our simple whiteboard model in figure 3.2 into a property graph model that we can use in the database.

3.2.1 The Property Graph model

We gave a brief overview of the property graph data model in chapter 1. Here we'll go through the process of taking our whiteboard model and converting it to a property graph model used by the database.

> **The Property Graph datamodel**
> The property graph model is composed of:
>
> - *Node Labels*: Nodes are the entities or objects in our data model. Nodes can have one or more labels that describe how nodes are grouped (think type or class).
> - *Relationships*: Relationships connect two nodes and have a single type and direction.
> - *Properties*: Arbitrary key-value pair attributes, stored on either nodes or relationships.

NODE LABELS

Nodes represent the objects in our whiteboard model. Each node can have one or more labels, which is a way of grouping nodes. Adding node labels (figure 3.3) to a whiteboard model is usually a simple process because a grouping will already have been defined during the whiteboard process. Here, we formalize the descriptors used to refer to our nodes into node labels (later we'll add node aliases and multiple labels so we use a colon as a separator to indicate the label).

Graph data model diagramming tools

There are many tools available for diagramming graph data models. Throughout this book we use the Arrows tool, a simple web-based application that allows for creating graph data models. Arrows is available online at http://www.apcjones.com/arrows/.

The Arrows user interface is minimal and is designed around creating property graph data models:

- Create new nodes with (+ Node).
- Drag relationships out of the halo of a node, either to an empty space for a new node or centered over an existing one to connect them.
- Double-click nodes and relationships to edit them and set names and properties (in a `key: "value" syntax).
- You can show the Markdown and also replace it with a previously saved fragment (useful for adding to documentation or checking into version control as your model evolves).
- You can export to SVG or take a screenshot.

For more on using Arrows, you can run the command `:play arrows` in Neo4j Browser to load a Neo4j Browser Guide showing how to use arrows or watch this quick video: bit.ly/33Nv9Mq.

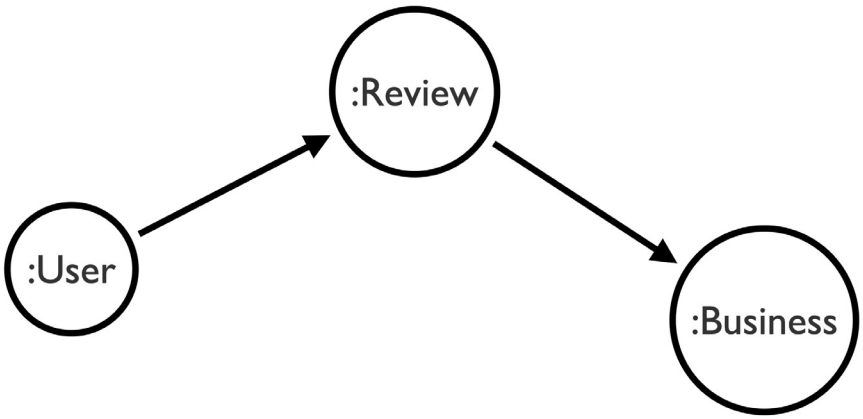


Figure 3.3 **Building the Property Graph model: node labels.**

The convention used for casing node labels is `PascalCase`. See the openCypher style guide for more examples of naming convention. Nodes can have multiple labels and allow us to represent type hierarchies, roles in different contexts, or even multi-tenancy.

Later on, we'll see how to use multiple node labels with GraphQL abstract types such as interface and union types.

RELATIONSHIPS

Once we've identified our nodes labels, the next step is to identify the relationships in our data model. Relationships have a single type and direction but can be queried in either direction (figure 3.4).

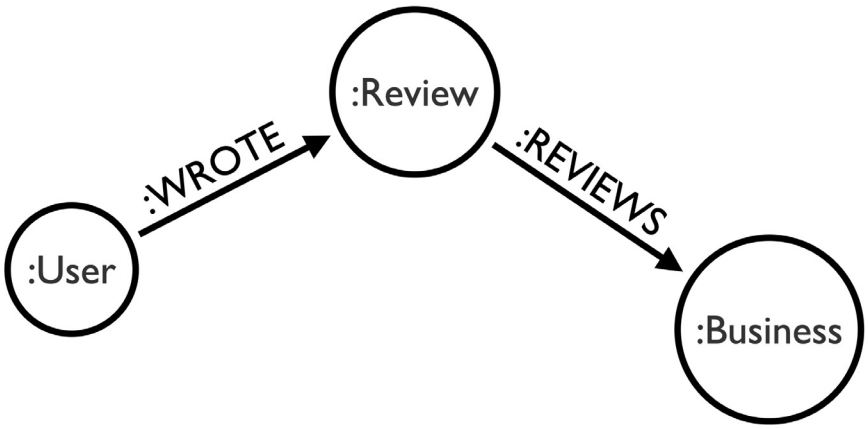


Figure 3.4 **Building the Property Graph model: relationship types.**

> **Dealing with undirected relationships**
> While every relationship has a single direction, we can treat the relationship as undirected at query time by not specifying a direction.

A good guideline for naming relationships is that the traversal from a node along a relationship to another node should read as a somewhat comprehensible sentence (figure 3.5). For example, "User wrote review" and "Review reviews business". You can read more about best practices for naming and conventions in the Cypher Style guide: https://neo4j.com/developer/cypher-style-guide

PROPERTIES

Properties are arbitrary key-value pairs stored on nodes and relationships. These are the attributes or fields of entities in our datamodel. Here we store `userId` and `name` as string properties on the User node, as well as other relevant properties on the Review and Business nodes.

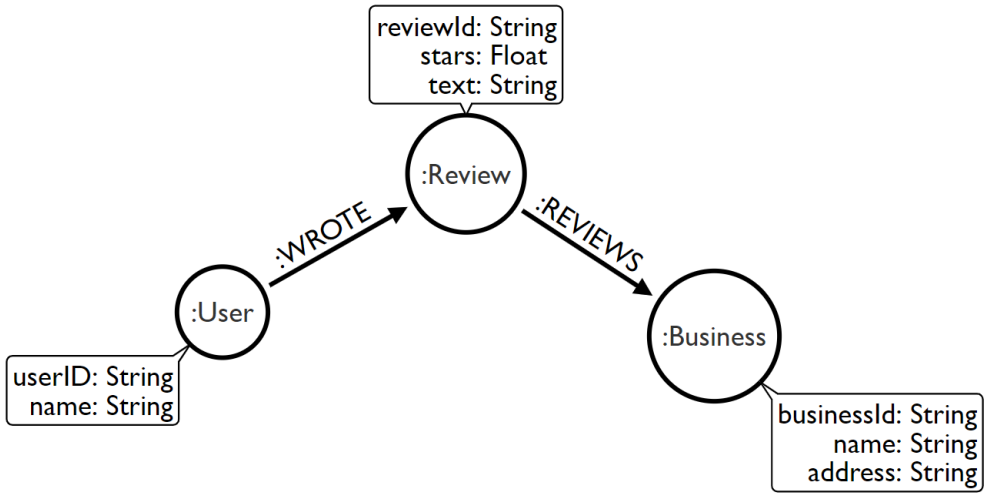


Figure 3.5 Building the Property Graph model: properties.

Property types

The following property types are supported by Neo4j:

- String
- Float
- Long
- Date
- DateTime
- LocalDateTime
- Time
- Point
- Lists of the above types

3.2.2 Database constraints and indexes

Now that we've defined our data model, how do we use it in the database? As mentioned earlier, unlike other databases that require us to define a complete schema before inserting data, Neo4j is said to be schema optional and doesn't require the use of a pre-defined schema. We can define database constraints that ensure the data adheres to the rules of the domain. We can create uniqueness constraints that ensure property values are unique across a node label (for example, guaranteeing that no two users have a duplicate id property value), property existence constraints (ensuring that a set of properties exist when a node or relationship is created or modified), and node key constraints that are similar to a composite key.

Database constraints are backed by indexes, which can be created separately as well. In a graph database indexes are used to find the starting point for a traversal, not

to traverse the graph. We'll cover database constraints and indexes in more detail in the following section introducing Cypher.

3.3 Data modeling considerations

Graph data modeling can be an iterative process. In general, this is the process followed:

1. What are the entities? How are they grouped? These become nodes and node labels.
2. How are these entities connected? These become relationships.
3. What are the attributes of the nodes and relationships? These become properties.
4. Can you identify the graph traversal that answers your questions? These become Cypher queries. If not, iterate on the graph model.

However, there are often nuances not covered by this general approach. In the following sections, we address several common graph data modeling questions.

3.3.1 Node vs. property

Sometimes it's difficult to determine if a value should be modeled as a node or as a property on the node. A good guideline to follow here is to ask yourself the question, "Can I discover something useful by traversing through it if it was a node?" If the answer is yes, then it should be modeled as a node; if not, then treat it as a property. For example, consider if we were to add the category of business to our model. Finding businesses with overlapping categories is potentially useful and easier to discover if category is modeled as a node.

3.3.2 Node vs. relationship

In the case where we have a piece of data that seemingly connects two nodes (such as a review of a business, written by a user), should we model this data as a node or as a relationship? At first glance it seems as if we might want to create a `REVIEWS` relationship connecting the user and business, storing the review information, such as `stars` and `text` as relationship properties. However, we might want to extract data from the review, such as keywords mentioned through some natural language processing technique and connect that extracted data to the review. Or perhaps we want to use the review nodes as the starting point for a traversal query? These are two examples of why we may want to choose to model this data as an intermediate node instead of a relationship.

3.3.3 Indexes

Indexes are used in graph databases to find the starting point of a traversal, and not during the actual traversal. This is an important performance characteristic of graph databases such as Neo4j known as index-free adjacency. Only create indexes for properties that will be used to find the starting point of a traversal, such as a username or business id.

3.3.4 Specificity of relationship types

Relationship types are a way of grouping relationships and should convey just enough information to make it clear how two nodes are connected without being overly specific. For example, `REVIEWS` is a good relationship type connecting `Review` and `Business` nodes, `REVIEW_WRITTEN_BY_BOB_FOR_PIZZA` is an overly specific relationship type, the name of the user and restaurant are stored elsewhere and don't need to be duplicated in the relationship type.

3.3.5 Choosing a relationship direction

All relationships in the property graph model have a single direction, but can be queried in either direction, or queried without consideration of direction. There's no need to create duplicate relationships to model bidirectionality. In general, you should choose relationship directions that allow for a consistent reading of the data model.

3.4 Tooling: Neo4j Desktop

Now that we understand the property graph data model and have defined a simple version of the model we'll use for our business reviews application, let's create a Neo4j database and start executing Cypher queries. To do this we'll use Neo4j Desktop, which is the mission control center for Neo4j. In Neo4j Desktop (figure 3.6), we can create projects and instances of Neo4j. We can start, stop, and configure Neo4j database instances in Neo4j Desktop, as well as install optional database plugins, such as GraphQL Algorithms, and APOC (a standard library of database procedures for

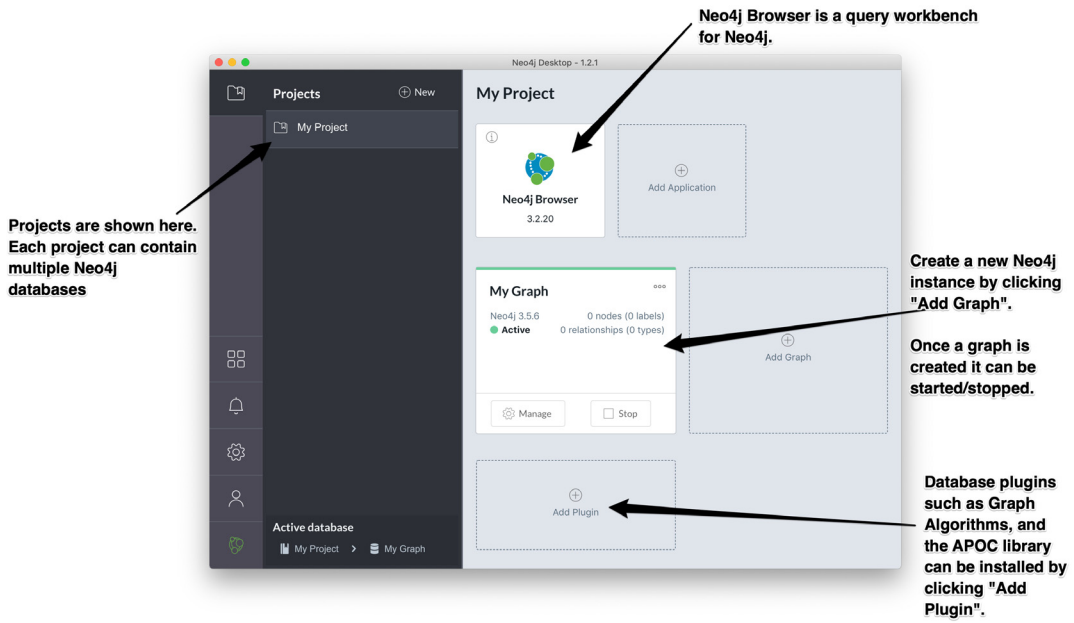


Figure 3.6 Neo4j Desktop: "Mission Control" for Neo4j.

Neo4j). Neo4j Desktop also includes functionality for installing "Graph Apps", which are applications that run in Neo4j Desktop and connect to the active Neo4j instance. Neo4j Browser, installed by default, is an example of one of these Graph Apps. See https://install.graphapp.io for examples of other Graph Apps.

If you haven't yet downloaded Neo4j Desktop do so now at https://neo4j.com/download. Neo4j Desktop is available to download for Mac, Windows, and Linux systems. Other options for downloading and running Neo4j, such as Docker, Debian package, or standalone Neo4j Server options can be found at neo4j.com/download-center although the remaining instructions will assume Neo4j Desktop is used.

Once you have downloaded and installed Neo4j, create a new local Neo4j instance by selecting Add Graph. You'll be prompted for a database name and password. The password can be anything you want, just be sure to remember it for later. Once you've created the graph, click the Start button to activate it, then we'll use Neo4j Browser to start querying the database we just created.

3.5 Tooling: Neo4j Browser

Neo4j Browser is a query workbench for Neo4j that allows developers to interact with the database by writing Cypher queries and visualizing the results (figure 3.7). Start Neo4j Browser by selecting its application icon in the Application section of Neo4j Browser.

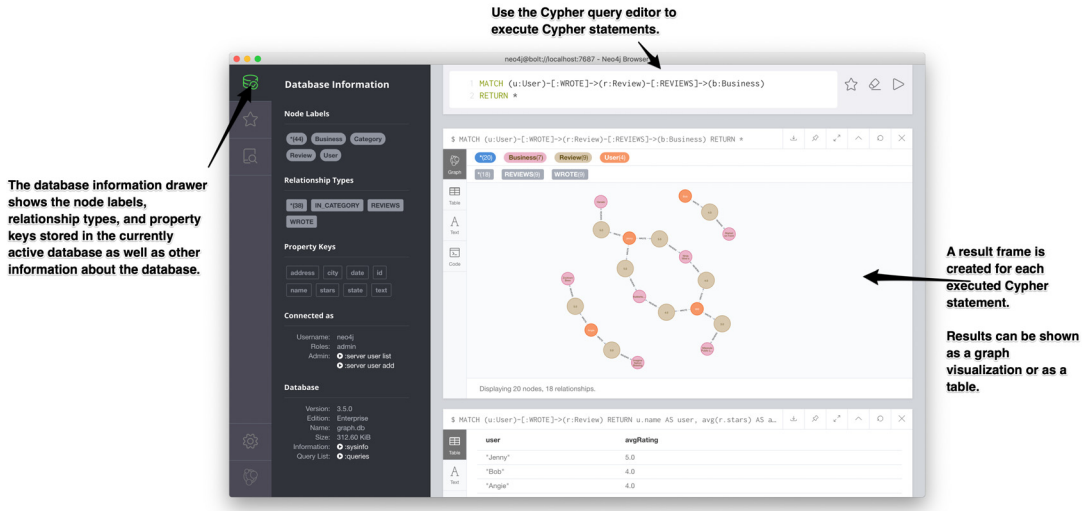


Figure 3.7 Neo4j Browser: A query workbench for Cypher and Neo4j.

Neo4j Browser allows us to run Cypher queries in Neo4j, but first let's review the Cypher query language.

3.6 Cypher

Cypher is a declarative graph query language with features that may be familiar from SQL. In fact, a good way to think of Cypher is as "SQL for graphs". Cypher uses pattern

matching, using an ASCII-art like notation for describing graph patterns. In this section, we'll look at basic Cypher functionality for creating and querying data, including making use of predicates and aggregations. We'll only cover a small part of the Cypher language. See the Cypher refcard https://r.neo4j.com/refcard for a through reference or consult the documentation at https://neo4j.com/docs/cypher-manual/current/ .

3.6.1 Pattern matching

As a declarative graph query language, pattern matching is a fundamental tool used in Cypher, both for creating and querying data. Instead of telling the database the exact operations, we want it to take (an imperative approach); with Cypher, we describe the pattern we're looking for or want to create and the database figures out the series of operations that satisfies the statement in the most efficient way possible. Describing graph patterns using an ASCII-art like notation (also called motifs) is at the heart of this declarative functionality.

NODES

Nodes are defined within parentheses `()`. Optionally, we can specify node label(s) using a colon as a separator, for example `(:User)`.

RELATIONSHIPS

Relationships are defined within square brackets `[]`. Optionally we can specify type and direction: `(:Review)-[:REVIEWS]→(:Business)`.

3.6.2 Properties

Properties are specified as comma-separated name: value pairs within braces "{}", like the name of a business or user.

ALIASES

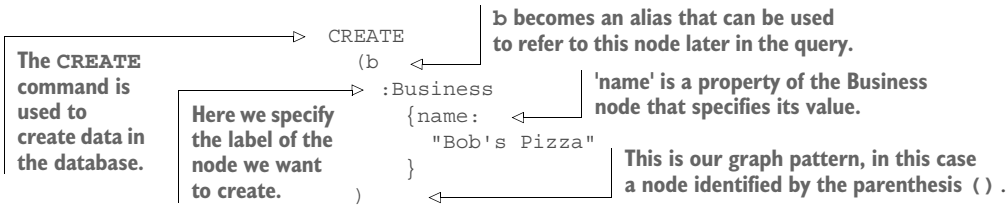
Graph elements can be bound to aliases that can be referred to later in the query: `(r:Review)-[a:REVIEWS]→(b:Business)`. The alias `r` becomes a variable bound to the review node matched in the pattern, `a` is bound to the `REVIEWS` relationship, and `b` is bound to the business node. These variables are only in scope for the Cypher query in which they're used.

Follow along by running the Cypher queries in Neo4j browser as we introduce Cypher commands for creating and querying data that matches the data model we built throughout this chapter.

3.6.3 CREATE

The first thing we need to do is create data in our database. Here's a look back at the objects we used in the previous chapter:

First, to create a single `Business` node in the graph, we start with the `CREATE` statement because we want to add data to the database. The `CREATE` clause takes a graph pattern upon which it operates:



And the result of running in Neo4j Browser shows:

```
Added 1 label, created 1 node, set 1 property, completed after 4 ms.
```

which means we've created 1 node with a new label in the database and set 1 node property value, in this case the `name` property on a node with the label `Business`.

Alternatively, we can use the `SET` command. This is equivalent:

```
CREATE (b:Business)
SET b.name = "Bob's Pizza"
```

To visualize the data being created we can add a `RETURN` clause to the Cypher statement, which will be rendered in Neo4j browser as a graph visualization. Running

```
CREATE (b:Business)
SET b.name = "Bob's Pizza"
RETURN b
```

gives the following visualization in Neo4j Browser (figure 3.8).

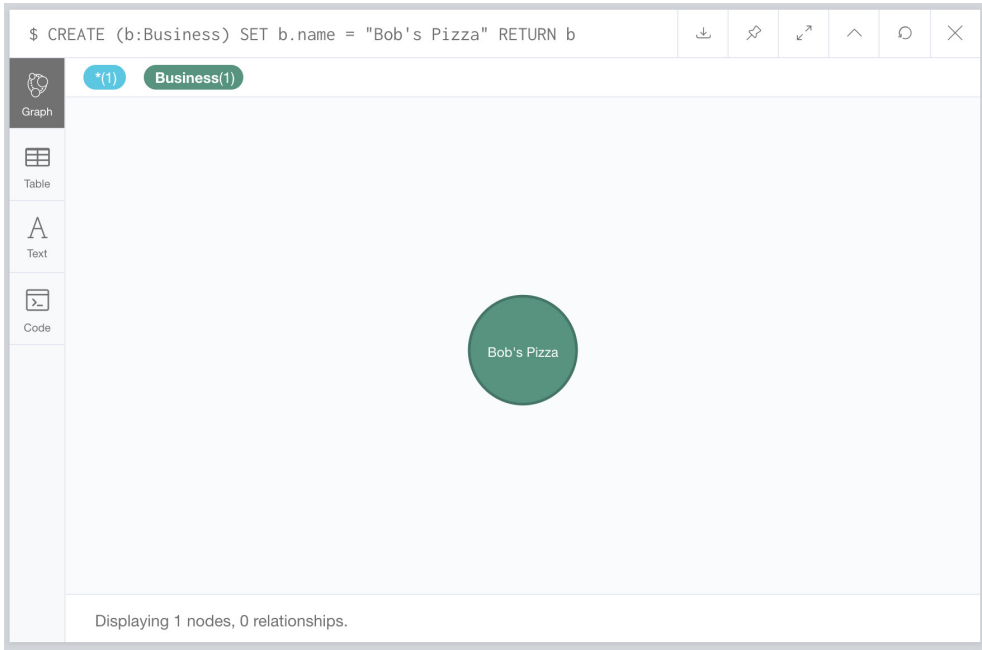


Figure 3.8 **Creating data with Cypher and Neo4j Browser.**

We can specify more complex patterns in the `CREATE` statement, such as relationships. Note the ASCII-art notation of defining a relationship using square brackets `←[]-`, including the direction of the relationship (figure 3.9).

```
CREATE (b:Business {name: "Bob's Pizza"})<-[:REVIEWS]-(r:Review {stars: 4,
     text: "Great pizza"})
RETURN b, r
```

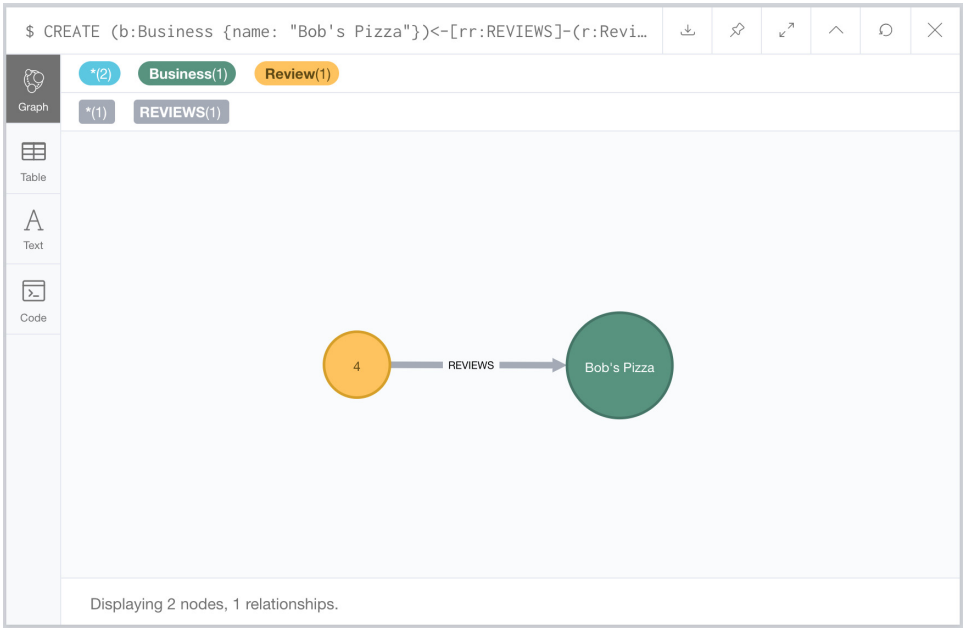


Figure 3.9 Creating data with Cypher and Neo4j Browser.

and even arbitrarily complex graph patterns (figure 3.10):

```
CREATE p=(b:Business {name: "Bob's Pizza"})<-[:REVIEWS]-(r:Review {stars: 4,
     text: "Great pizza"})<-[:WROTE]-(u:User {name: "Willie"})
RETURN p
```

Note that in the previous Cypher query we bind the entire graph pattern to a variable `p` and return that variable. In this case, `p` takes on the value of the entire path (a combination of nodes and relationships) being created.

Up to now we've returned only the data we've created in each Cypher statement. How do we query and visualize the rest of the data in the database? To do this, we use the `MATCH` keyword. Let's match on all nodes in the database and return them:

```
MATCH (a) RETURN a
```

and we should see a graph that looks something like figure 3.11.

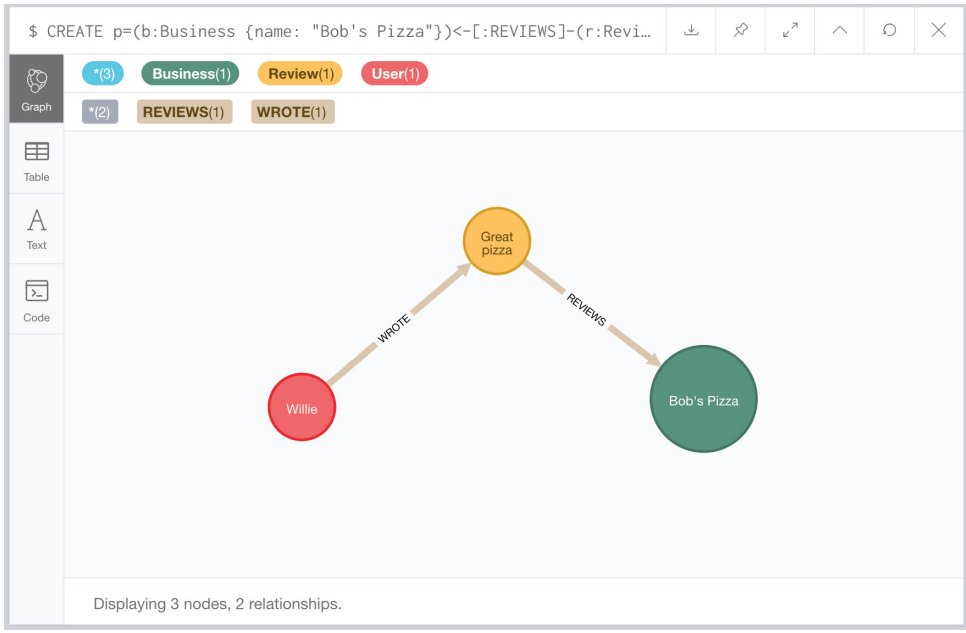


Figure 3.10 Creating data with Cypher and Neo4j Browser.

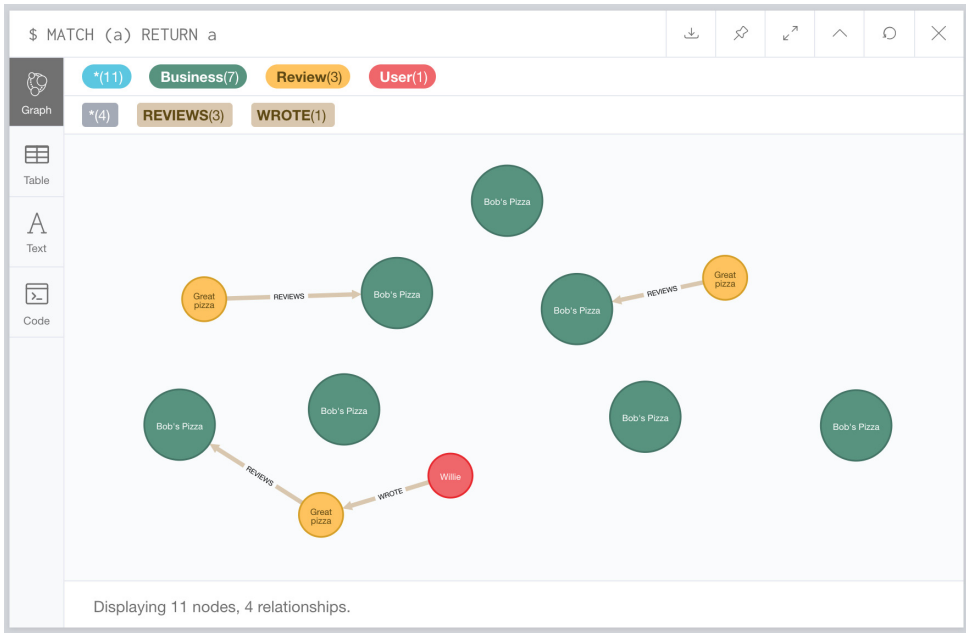


Figure 3.11 Creating data with Cypher and Neo4j Browser.

Right away, we can see something is wrong; we've created many duplicate nodes in our graph! Let's delete all data in the database:

```
MATCH (a) DETACH DELETE a
```

This will match on all nodes and delete both the nodes and any relationships.

We should see output that tells us what we've deleted:

```
Deleted 11 nodes, deleted 4 relationships, completed after 23 ms.
```

Now let's start over and learn how to create data in the database without creating duplicates.

3.6.4 MERGE

To avoid creating duplicates, we can use the `MERGE` command. `MERGE` acts as an upset: only creating data specified in the pattern if it doesn't already exist in the database. When using `MERGE`, it's best to create a uniqueness constraint on the property that identifies uniqueness—often an id field. By creating a uniqueness constraint, this will also create an index in the database. See the next section for example of creating uniqueness constraints. For simple examples, it's fine to use `MERGE` without these constraints, so let's revisit our Cypher statement that created a business, a review and user, but this time we'll use `MERGE`:

```
MERGE (b:Business {name: "Bob's Pizza"})
MERGE (r:Review {stars: 4, text: "Great pizza!"})
MERGE (u:User {name: "Willie"})
MERGE (b)<-[:REVIEWS]-(r)<-[:WROTE]-(u)
RETURN *
```

And the resulting graph visualization, showing the data we've created in figure 3.12.

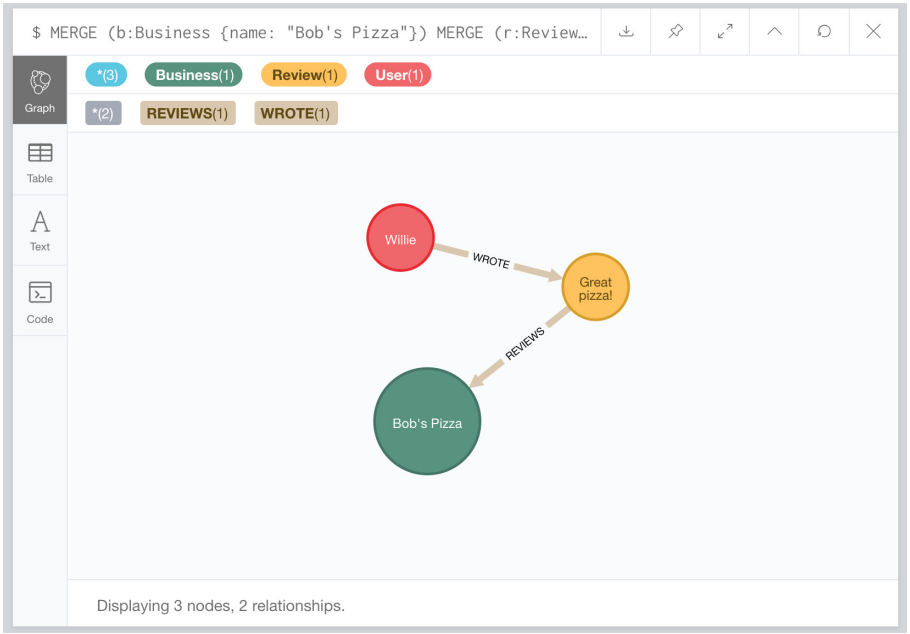


Figure 3.12 Using MERGE to create data.

The results of this Cypher statement look identical to the version using CREATE; however, there's an important difference: this query is now idempotent. No matter how many times we run the query, we won't create duplicate nodes, because we're using MERGE instead of CREATE. We'll revisit MERGE again in the next chapter when we show how to create data in the database via our GraphQL API.

Indexes In Neo4j

It's important to understand how indexes are used in a graph database like Neo4j. We said earlier that Neo4j has a property called index-free adjacency, which means that traversing from a node to any other connected node does not require an index lookup. How are indexes used in Neo4j? Indexes are used to find the starting point for a traversal only, unlike relational databases that use an index to compute set (table) overlap, graph databases are simply computing offsets in the file store, essentially chasing pointers, which we know computers are good at doing quickly.

3.6.5 Defining database constraints with Cypher

We mentioned database constraints and how they relate to (optionally) defining a schema in Neo4j earlier in the chapter as we built up our data model. Here we show the Cypher syntax for creating database constraints relevant to our data model.

Uniqueness constraints are used to assert that the database will never contain more than one node with a specific label and property value. In this case we create a uniqueness constraint ensuring that the value of the businessId property on nodes with the label Business is unique.

```
CREATE CONSTRAINT ON (b:Business) ASSERT b.businessId IS UNIQUE;
```

Property existence constraints ensure that all nodes with a certain label have a certain property. Here we create a constraint to guarantee that all Business nodes will have a name property, however the value for this property does not need to be unique across all Business nodes.

```
CREATE CONSTRAINT ON (b:Business) ASSERT exists(b.name);
```

Node key constraints ensure that all nodes with a certain label have a set of defined properties whose combined value is unique and that all properties in this set exist on the node. Here we create a node key constraint ensuring that the combination of firstName and lastName is unique for Person nodes, and that every Person node has both firstName and lastName properties.

```
CREATE CONSTRAINT ON (p:Person) ASSERT (p.firstName, p.lastName) IS NODE KEY;
```

Note that if you still have duplicate data in the database that conflict with any of these constraints, then you'll receive an error message saying the constraint cannot be created.

3.6.6 MATCH

Now that we've created our data in the graph, we can start to write queries to address several of the business requirements of our application.

The MATCH clause is similar to CREATE in that it takes a graph pattern; however, we can also use a WHERE clause for specifying predicates to be applied in the pattern.

```
MATCH (u:User)
RETURN u
```

We can, of course, use more complex graph patterns in a MATCH clause:

```
MATCH (u:User)-[:WROTE]->(r:Review)-[:REVIEWS]->(b:Business)
RETURN u, r, b
```

The previous query matches on all users that have written a review of any business. What if instead we only want to query for reviews of a certain business? In that case, we need to introduce predicates into our query. Notice how the relationship arrows are reversed here.

WHERE

The WHERE clause can be used to add predicates to a MATCH statement. To search for the business named "Bob's Pizza":

```
MATCH (b:Business)
WHERE b.name = "Bob's Pizza"
RETURN b
```

For equality comparisons, an equivalent shorthand notation is available:

```
MATCH (b:Business {name: "Bob's Pizza"})
RETURN b
```

3.6.7 Aggregations

Often, we want to compute an aggregation across a set of results. For example, we may want to calculate the average rating of all the reviews of Bob's Pizza. To do this, we use the avg aggregation function:

```
MATCH (b:Business {name: "Bob's Pizza"})<-[:REVIEWS]-(r:Review)
RETURN avg(r.stars)
```

Now in Neo4j Browser because we're not returning graph data, and rather tabular data, instead of a graph visualization, we're presented with a table showing the results of our query:

```
╒═══════════════╕
│"avg(r.stars)"│
╞═══════════════╡
│4.0            │
└───────────────┘
```

What if we wanted to calculate the average rating of *each* business? In SQL, we might use a GROUP BY operator to group the reviews by business name and calculate the

aggregation across each group, but there's no `GROUP BY` operator in Cypher. Instead, with Cypher there's an *implicit* group by when returning the results of an aggregation function along with non-aggregated results. For example, to compute the average rating of each business:

```
MATCH (b:Business)<-[:REVIEWS]-(r:Review)
RETURN b.name, avg(r.stars)
```

and the results table:

"b.name"	"avg(r.stars)"
"Bob's Pizza"	4.0

Of course, this isn't too exciting because we only have one business and one review. In the exercise section of this chapter, we'll work with a larger dataset.

3.7 Using the client drivers

Until now we've used Neo4j Browser to execute our Cypher queries, which is usual for ad-hoc analysis or prototyping; however, typically we want to create an application that interacts with the database. To do this, we use the Neo4j client drivers. These client drivers are available in many languages such as JavaScript, Java, Python, .NET, and Go, and allow the developer to execute Cypher queries against a Neo4j instance with a consistent API that is idiomatic to the language. In chapter 1 we saw an example of using the Neo4j JavaScript driver to execute a Cypher query and work with the results. Neo4j client drivers also provide a fundamental building block for building framework-specific integrations with Neo4j, such as neo4j-graphql.js, which we'll explore in future chapters. Refer to the Drivers & Language guides for more information on Neo4j client drivers: https://neo4j.com/developer/language-guides/ .

In the next chapter, we'll see how to build a GraphQL API that connects to Neo4j using the Neo4j JavaScript driver and the neo4j-graphql.js library.

Exercises

To complete the following exercises, first run the following command in Neo4j Browser: `:play grandstack` to load a browser guide with embedded queries. This browser guide will walk you through the process of loading a larger, more complete sample dataset of businesses and reviews. After running the query to load the data in Neo4j:

1. Run the command `CALL db.schema()` to inspect the data model. What are the node labels used? What are the relationship types?
2. Write a Cypher query to find all the users in the database. How many users are there? What are their names?

3. Find all the reviews written by the user named "Will". What's the average rating given by this user?
4. Find all the businesses reviewed by the user named "Will". What's the most common category?
5. Write a query to recommend businesses to the user named "Will" that he hasn't previously reviewed.

You can find solutions to the exercises, as well as code samples, in the GitHub repository for this book: https://github.com/johnymontana/fullstack-graphql-book.

Summary

- A graph database allows you to model, store, and query data as a graph.
- The property graph data model is used by graph databases and consists of node labels, relationships, and properties.
- The Cypher query language is a declarative graph query language focused around pattern matching and is used for querying graph databases, including Neo4j.
- Client drivers are used for building applications that interact with Neo4j. These drivers enable application to send Cypher queries to the database and work with the results.

4 A GraphQL API for our graph database

This chapter covers

- Reviewing common issues that arise when building GraphQL backends
- Introducing database integrations for GraphQL that address common problems
- Building a GraphQL endpoint backed by Neo4j
- Extending the functionality of our GraphQL API with custom logic
- Inferring a GraphQL endpoint from an existing Neo4j database

GraphQL backend implementations commonly run into a set of issues that negatively impact performance and developer productivity. We've identified several of these problems previously (such as the "n+1 query problem"), and in this chapter we look deeper at the common issues that arise and discuss how they can be mitigated using database integrations for GraphQL that make it easier to build efficient GraphQL APIs backed by databases.

Specifically, we look at using neo4j-graphql.js, a Node.js library designed to work with JavaScript GraphQL implementations such as Apollo Server for building GraphQL APIs backed by Neo4j. neo4j-graphql.js allows us to generate GraphQL APIs from type definitions, driving the database data model from GraphQL, auto-generate resolvers for data fetching and mutations, including complex filtering, ordering, and pagination. neo4j-graphql.js also enables adding custom logic.

In this chapter, we look at using neo4j-graphql.js to integrate our business reviews GraphQL API with Neo4j, adding a persistence layer to our API. In this initial look at neo4j-graphql.js, we focus on querying existing data, using the sample dataset in Neo4j used in the previous chapter. We'll explore creating and updating data (GraphQL mutations), as well as more complex GraphQL querying semantics, such as interfaces and fragments in future chapters, introducing these concepts in the context of building out our user interface. Figure 4.1 shows how neo4j-graphql.js fits into the larger architecture of our application. The goal of neo4j-graphql.js is to make it easy to build an API backed by Neo4j.

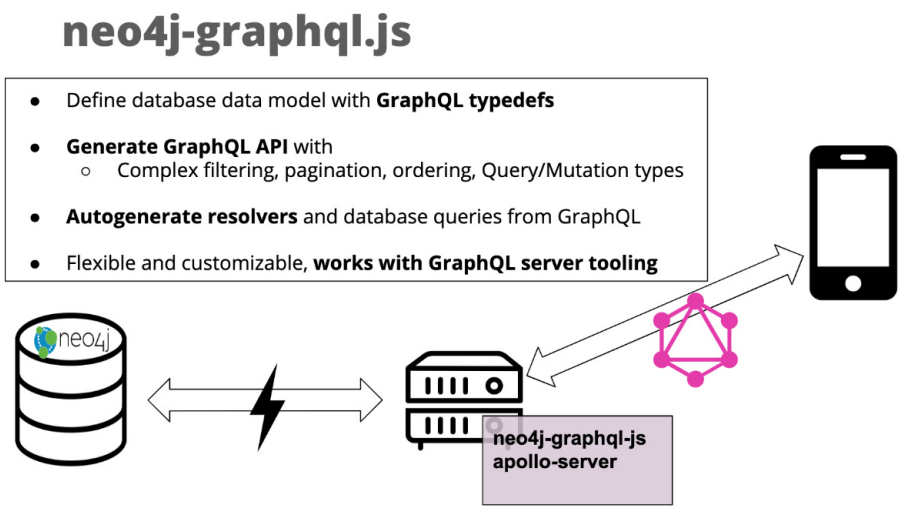


Figure 4.1 neo4j-graphql.js helps build the API layer between client and database.

4.1 Common GraphQL problems

When building GraphQL APIs, two types of common problems occur that developers can face: poor performance and writing lots of boilerplate code that can impact developer productivity.

4.1.1 Poor performance and the N+1 query problem

We've described the N+1 query problem previously. Because of the nested way that GraphQL resolvers are called, multiple database requests are often required to resolve a GraphQL query from the data layer. For example, imagine a query searching for

businesses by name as well as all reviews for each business. A naive implementation would first query the database for all businesses matching the search phrase. Then for each matching business they send an additional query to the database to find any reviews for the business. Each query to the database incurs network and query latency, which can significantly impact performance.

A common solution for this is to use a caching and batching pattern known as DataLoader. This can alleviate part of the performance issues; however, it can still require multiple database requests and cannot be used in all cases, such as when the ID of an object isn't known.

4.1.2 Boilerplate and developer productivity

The term boilerplate is used to describe repetitive code that's written to accomplish a common task. In the case of implementing GraphQL APIs, often writing boilerplate code to implement data fetching logic in resolvers is required. This can negatively impact developer productivity, slowing down development as the developer is required to write simple data fetching logic for each type and field instead of focusing on the key components of their application. In the context of our business reviews application, this means manually writing the logic for finding businesses by name in the database, as well as finding reviews associated with each business and each user connected to each review, and so on, until we've manually defined the logic for fetching all fields of our GraphQL schema.

4.2 Introducing GraphQL database integrations

GraphQL database integrations are a class of tools that enable building GraphQL APIs that interact with databases. A handful of these tools exist with different feature sets and levels of integration—in this book we focus on neo4j-graphql.js. However, in general, the goal of these GraphQL "engines" is to address the common GraphQL problems identified previously in a consistent way by reducing boilerplate and addressing data fetching performance.

Throughout the rest of this chapter, we focus on using neo4j-graphql.js to build a GraphQL API backed by Neo4j. It's important to note that our GraphQL API serves as a layer between the client and the database; we don't want to directly query our database from the client. The API layer serves an important function where we can implement features such as authorization and custom logic that we don't want to expose to the client. Also, because GraphQL is an API query language (not a database query language), it lacks many semantics (such as aggregations and projections) that we expect in a database query language.

4.3 The neo4j-graphql.js library

Neo4j-graphql.js is a Node.js library that works with any JavaScript GraphQL implementation, such as GraphQL.js and Apollo Server, designed to make it as easy as possible to build GraphQL APIs backed by a Neo4j database. The two main functions of neo4j-graphql.js are *schema augmentation* and *GraphQL transpilation* (figure 4.2).

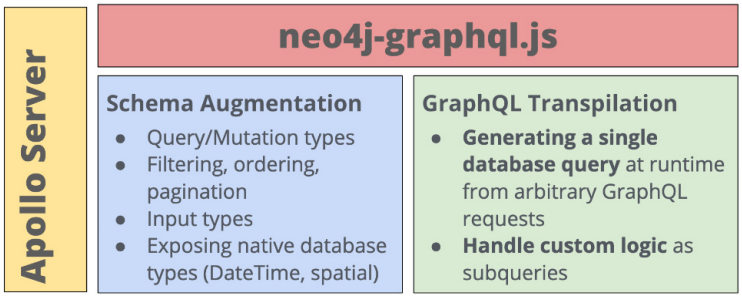


Figure 4.2 The two main functions of neo4j-graphql.js.

The schema augmentation process takes GraphQL type definitions and generates a GraphQL API with CRUD (Create, Read, Update, Delete) operations for the types defined. In GraphQL semantics, this includes adding a `Query` and `Mutation` type to the schema and generating resolvers for these queries and mutations. The generated API includes support for filtering, ordering, pagination, and native database types such as spatial and temporal types, without having to define these manually in the type definitions. The result of this process is a GraphQL executable schema object that can then be passed to a GraphQL server implementation, such as Apollo Server, to serve the API and handle networking and GraphQL execution processes. The schema augmentation process eliminates the need to write boilerplate code for data fetching and mapping the GraphQL and database schemas.

The GraphQL transpilation process happens at query time. When a GraphQL request is received, a single Cypher query is generated that can resolve the request and is sent to the database. Generating a single database query solves the n+1 query problem, assuring only one round trip to the database per GraphQL request.

You can find the documentation for neo4j-graphql.js at https://grandstack.io/docs.

4.3.1 Project setup

Throughout the rest of the chapter, we'll explore the features of neo4j-graphql.js by creating a new GraphQL API for Neo4j that uses the sample dataset of businesses and reviews from the Exercise section of the previous chapter. We'll first create a new Node.js project that uses neo4j-graphql.js and the Neo4j JavaScript driver to fetch data from Neo4j. Then we'll explore the various features of neo4j-graphql.js, adding additional code as we move along.

NEO4J

First, make sure a Neo4j instance is running (you can use Neo4j Desktop, Neo4j Sandbox, or Neo4j Aura, but we'll assume Neo4j Desktop for the purposes of this chapter). If using Neo4j Desktop, you need to install the APOC standard library plugin. Don't worry about this step if using Neo4j Sandbox or Neo4j Aura, because APOC is included

by default in those services. To install APOC in Neo4j Desktop, click the "Plugins" tab in your project, then look for APOC in the list of available plugins, and click "Install".

Next, make sure your Neo4j database is empty by running the Cypher statement in listing 4.1.

CAUTION This statement will delete all data in your Neo4j database so make sure this is the instance you want to use and not a database you don't want to delete.

Listing 4.1 Clearing out our Neo4j database

```
MATCH (a) DETACH DELETE a;
```

Now we're ready to load our sample dataset (figure 4.3), which you may have done already if you completed the exercise section in the previous chapter. Run the following command in Neo4j Browser:

```
:play grandstack
```

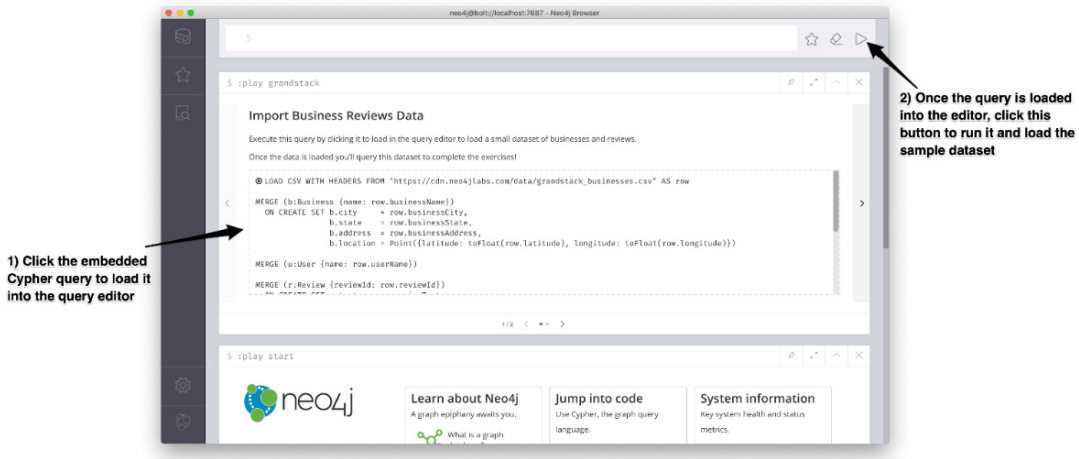


Figure 4.3 Loading the sample dataset into Neo4j.

This will load a sample dataset into Neo4j that we'll use as the basis for our GraphQL API. Next, we can explore the data a bit by running a command in the following listing that will give us a visual overview of the data included in the sample dataset (figure 4.4).

Listing 4.2 Visualizing the graph schema in Neo4j

```
CALL db.schema.visualization();
```

We see that we have four node labels: `Business`, `Review`, `Category`, and `User` connected by three relationship types: `IN_CATEGORY` (connecting businesses to the categories to which they belong), `REVIEWS` (connecting reviews to businesses), and `WROTE` (connecting users to reviews they have authored).

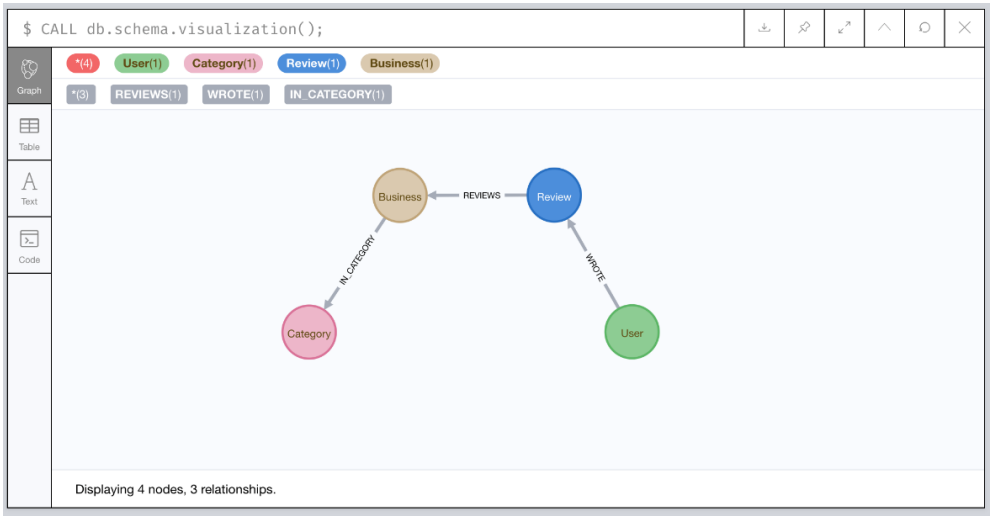


Figure 4.4 **The graph schema of our sample dataset.**

We can also view the node properties stored on the various node labels.

Listing 4.3 **Inspect the node properties stored in Neo4j**

```
CALL db.schema.nodeTypeProperties()
```

This command renders a table showing us the property names, their types and whether or not they're found on all nodes of that label.

"nodeType"	"nodeLabels"	"propertyName"	"propertyTypes"	"mandatory"
":`User`"	["User"]	"name"	["String"]	true
":`User`"	["User"]	"userId"	["String"]	true
":`Review`"	["Review"]	"reviewId"	["String"]	true
":`Review`"	["Review"]	"text"	["String"]	false
":`Review`"	["Review"]	"stars"	["Double"]	true
":`Review`"	["Review"]	"date"	["Date"]	true
":`Category`"	["Category"]	"name"	["String"]	true
":`Business`"	["Business"]	"name"	["String"]	true
":`Business`"	["Business"]	"city"	["String"]	true

":`Business`"	["Business"]	"state"	["String"]	true
":`Business`"	["Business"]	"address"	["String"]	true
":`Business`"	["Business"]	"location"	["Point"]	true
":`Business`"	["Business"]	"businessId"	["String"]	true

We'll use this table in a few moments when we construct GraphQL type definitions that describes this graph.

NODE.JS APP

Now that we have our Neo4j database loaded with our sample dataset let's set up a new node.js project for our GraphQL API:

```
npm init -y
```

And install our dependencies:

- *neo4j*-graphql.js: A package that makes it easier to use GraphQL and Neo4j together. neo4j-graphql-js translates GraphQL queries to a single Cypher query, eliminating the need to write queries in GraphQL resolvers and for batching queries. It also exposes the Cypher query language through GraphQL via the `@cypher` schema directive.
- apollo-server: Apollo Server is an open-source GraphQL server that works with any GraphQL schema built with graphql.js, including neo4j-graphql.js. It also has options for working with many different Node.js webserver frameworks, or the default Express.js.
- neo4j-driver: The Neo4j drivers allow for connecting to a Neo4j instance, either local or remote, and executing Cypher queries over the Bolt protocol. Neo4j drivers are available in many different languages and here we use the Neo4j JavaScript driver.

```
npm install neo4j-graphql-js apollo-server neo4j-driver
```

Now create a new file `index.js` and let's add initial code in the following listing.

Listing 4.4 index.js: Initial GraphQL API code

```
const { ApolloServer } = require("apollo-server");
const neo4j = require("neo4j-driver");
const { makeAugmentedSchema } = require("neo4j-graphql-js");

const typeDefs = /* GraphQL */ ``;

const schema = makeAugmentedSchema({
  typeDefs
});
```

Here we pull in our dependencies.

This line serves as a placeholder for our GraphQL type definitions to be filled in later.

makeAugmentedSchema generates resolvers for our type definitions.

Here we start the GraphQL server.

Our generated GraphQL schema is passed to Apollo Server.

```
const server = new ApolloServer({
  schema
});

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

This is the basic structure for our GraphQL API application code. We can run it on the command line using the `node` command:

```
node index.js
```

However, we'll quickly see an error message complaining that we haven't provided GraphQL type definitions.

```
➜ node index.js
/Users/lyonwj/api/node_modules/neo4j-graphql-js/dist/index.js:293
  if (!typeDefs) throw new Error('Must provide typeDefs');
  ...
```

We must provide either GraphQL type definitions or a GraphQL schema object to `makeAugmentedSchema` that defines the GraphQL API, so the next step is to fill in our GraphQL type definitions.

4.3.2 Generated GraphQL schema from type definitions

Following the GraphQL-First approach described previously, our GraphQL type definitions drive the API specification. In this case, we know what data we want to expose (our sample dataset loaded in Neo4j), so we can refer to the table of node properties above and apply a simple rule as we create our GraphQL type definitions: node labels become types, taking on the node properties as fields. We also need to define relationship fields in our GraphQL type definitions. Let's first look at the complete type definitions and then explore how we define relationship fields in the following listing.

Listing 4.5 index.js: GraphQL type definitions

```
const typeDefs = /* GraphQL */ `
  type Business {
    businessId: ID!
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
  }

  type User {
    userID: ID!
    name: String!
    reviews: [Review] @relation(name: "WROTE", direction: "OUT")
  }
```

```
  type Review {
    reviewId: ID!
    stars: Float!
    date: Date!
    text: String
    user: User @relation(name:"WROTE", direction: "IN")
    business: Business @relation(name: "REVIEWS", direction: "OUT")
  }

  type Category {
    name: String!
    businesses: [Business] @relation(name: "IN_CATEGORY", direction: "IN")
  }
`;
```

@RELATION GRAPHQL SCHEMA DIRECTIVE

In the property graph model used by Neo4j, every relationship has a direction and type. To represent this in GraphQL, we use GraphQL schema directives, specifically the `@relation` schema directive. A directive is like an annotation in our GraphQL type definitions. It's an identifier preceded by the `@` character, which may then optionally contain a list of named arguments. Schema directives are GraphQL's built-in extension mechanism, indicating some custom logic on the server.

When defining relationship fields using the `@relation` directive, the `name` argument indicates the relationship type stored in Neo4j and the `direction` argument indicates the relationship direction.

In addition to schema directives, directives can also be used in GraphQL queries to indicate specific behavior. We'll see examples of query directives when we explore GraphQL clients.

4.3.3 Generated data fetching

The autogenerated resolvers in our GraphQL schema need to access our Neo4j database using the Neo4j JavaScript driver and they expect a driver instance to be injected into the context object that's passed to each resolver. By convention, the driver is injected under the key `driver` in the context object, as shown in the following listing.

Listing 4.6 index.js: Creating a Neo4j driver instance

```
const driver = neo4j.driver(
  "bolt://localhost:7687",
  neo4j.auth.basic("neo4j", "letmein")
);

const server = new ApolloServer({
  schema,
  context: { driver }
});
```

Creating a Neo4j driver instance using credentials for our Neo4j database.

We inject this driver instance into the context object.

4.3.4 Configuring the generated API

We mentioned that the schema augmentation process adds queries and mutations for each type defined in the type definitions. We can configure the generated API, disabling queries or mutations altogether, or specify that certain types be excluded.

DISABLING AUTO-GENERATED QUERIES AND MUTATIONS

Because we're initially only focusing on the query API, let's disable all generated mutations in the following listing.

Listing 4.7 index.js: Disabling mutations

```
const schema = makeAugmentedSchema({
  typeDefs,
  resolvers,
  config: {
      mutation: false
  }
});
```

EXCLUDING TYPES

We can also pass an array of types to be excluded from the generated queries or mutations, as shown in the following listing.

Listing 4.8 index.js: Disabling specific types

```
const schema = makeAugmentedSchema({
  typeDefs,
  resolvers,
  config: {
      mutation: false,
      query: {
          exclude: ["MySecretType"]
      }
  }
});
```

Now, let's run our API application:

```
node index.js
```

As output, we should see the address where our API application is listening, in this case on port 4000 on `localhost`.

```
➜  node index.js
GraphQL server ready at http://localhost:4000/
```

Navigate to http://localhost:4000 in your web browser and you should see the familiar GraphQL Playground interface. Open the "Docs" tab in GraphQL to see the generated API (figure 4.5). Spend a few minutes looking through the Query field descriptions, and you'll notice arguments have been added to types for things such as ordering, pagination, and filtering.

4.4 Basic queries

Now that we have our GraphQL server powered by Apollo Server and neo4j-graphql.js up and running, let's start querying our API using GraphQL Playground. Looking at the `Docs` tab in GraphQL Playground, we can see the API entrypoints (in GraphQL

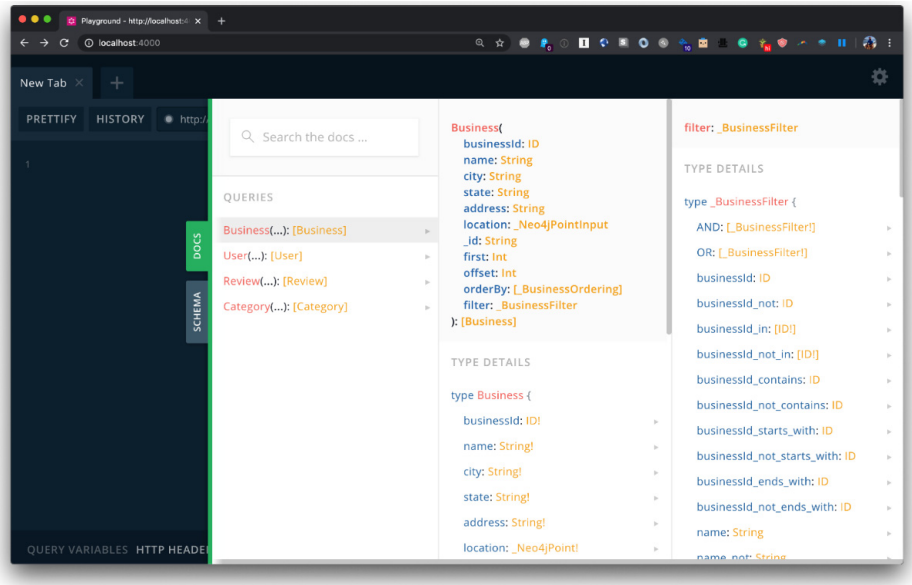


Figure 4.5 **GraphQL Playground showing our generated API.**

parlance, each Query type field is an entry point to the API) available to us: `Business`, `User`, `Review`, `Category`, one for each type defined in our type definitions.

Let's start by querying for all businesses and return only the name field, as shown in the following listing.

Listing 4.9 GraphQL query

```
{
  Business {
    name
  }
}
```

If we run this query in GraphQL Playground, we should see the following results listing businesses and their names only.

```
{
  "data": {
    "Business": [
      {
        "name": "Missoula Public Library"
      },
      {
        "name": "Ninja Mike's"
      },
      {
        "name": "KettleHouse Brewing Co."
      },
```

```
      {
        "name": "Imagine Nation Brewing"
      },
      {
        "name": "Market on Front"
      },
      {
        "name": "Hanabi"
      },
      {
        "name": "Zootown Brew"
      },
      {
        "name": "Ducky's Car Wash"
      },
      {
        "name": "Neo4j"
      }
    ]
  }
}
```

Neat! This data has been fetched from our Neo4j instance for us, and we didn't even need to write any resolvers!

If we check the console output in the terminal, we can see the generated Cypher query logged to the terminal in the following listing.

Listing 4.10 Generated Cypher query

```
MATCH (`business`:`Business`) RETURN `business` { .name } AS `business`
```

We can add additional fields to the GraphQL query and those fields will be added to the generated Cypher query, returning only the data needed.

For example, the following GraphQL query adds the address of the business as well as the name field, as shown in the following listing.

Listing 4.11 GraphQL query

```
{
  Business {
    name
    address
  }
}
```

The Cypher translation of the GraphQL query also now includes the address field, as shown in the following listing.

Listing 4.12 Generated Cypher query

```
MATCH (`business`:`Business`) RETURN `business` { .name , .address } AS
     `business`
```

And finally, when we examine the results of the GraphQL query, we now see an address listed for each business.

```
{
  "data": {
    "Business": [
      {
        "name": "Missoula Public Library",
        "address": "301 E Main St"
      },
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St"
      },
      {
        "name": "KettleHouse Brewing Co.",
        "address": "313 N 1st St W"
      },
      {
        "name": "Imagine Nation Brewing",
        "address": "1151 W Broadway St"
      },
      {
        "name": "Market on Front",
        "address": "201 E Front St"
      },
      {
        "name": "Hanabi",
        "address": "723 California Dr"
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St"
      },
      {
        "name": "Ducky's Car Wash",
        "address": "716 N San Mateo Dr"
      },
      {
        "name": "Neo4j",
        "address": "111 E 5th Ave"
      }
    ]
  }
}
```

Next, let's take advantage of several of the features of the generated GraphQL API.

4.5 Ordering and pagination

Each input type field includes `first`, `offset` and `orderBy` arguments to enable ordering and pagination. In the next listing, we search for the first three businesses, ordered by the value of the `name` field.

Listing 4.13 Initial GraphQL API code

```
{
  Business(first: 3, orderBy: name_asc) {
    name
  }
}
```

Ordering enums are generated for each type, offering ascending and descending options for each field. For example, in the next listing we see the ordering enum generated for the `Business` type.

Listing 4.14 Initial GraphQL API code

```
enum _BusinessOrdering {
  businessId_asc
  businessId_desc
  name_asc
  name_desc
  city_asc
  city_desc
  state_asc
  state_desc
  address_asc
  address_desc
  _id_asc
  _id_desc
}
```

Running our query returns businesses now ordered by name, as shown in the following listing.

Listing 4.15 Initial GraphQL API code

```
{
  "data": {
    "Business": [
      {
        "name": "Ducky's Car Wash"
      },
      {
        "name": "Hanabi"
      },
      {
        "name": "Imagine Nation Brewing"
      }
    ]
  }
}
```

If we switch to the terminal, we can see the Cypher query generated from our GraphQL query, which now includes `ORDER BY` and `LIMIT` clauses that map to our `first` and `orderBy` GraphQL arguments. The ordering and limiting is executed in

the database, rather than in the client, so only the necessary data is returned from the database query. See the following listing.

Listing 4.16 Generated Cypher query

```
MATCH (`business`:`Business`) WITH `business` ORDER BY business.name ASC
RETURN `business` { .name } AS `business` LIMIT toInteger($first)
```

Note that this query includes a `$first` parameter, rather than including the value `3` inline in the query. Parameter usage is important here because it ensures a user can't inject potentially malicious Cypher code into the generated query and also it ensures the query plan generated by Neo4j can be reused, increasing performance.

To run this query in Neo4j Browser first set a value for the `first` parameter with the `:param` command:

```
:param first => 3
```

4.6 Nested queries

Cypher can easily express the types of graph traversals in our GraphQL queries, and neo4j-graphql.js is capable of generating the equivalent Cypher queries for arbitrary GraphQL requests, including nested queries.

In the following listing, we traverse from businesses to their categories.

Listing 4.17 GraphQL query

```
{
  Business(first: 3, orderBy: name_asc) {
    name
    categories {
      name
    }
  }
}
```

And the result shows each business is connected to one or more categories.

```
{
  "data": {
    "Business": [
      {
        "name": "Ducky's Car Wash",
        "categories": [
          {
            "name": "Car Wash"
          }
        ]
      },
      {
        "name": "Hanabi",
        "categories": [
          {
            "name": "Restaurant"
          },
```

```
          {
            "name": "Ramen"
          }
        ]
      },
      {
        "name": "Imagine Nation Brewing",
        "categories": [
          {
            "name": "Beer"
          },
          {
            "name": "Brewery"
          }
        ]
      }
    ]
  }
}
```


4.7 Filtering

The filter functionality is exposed by adding a filter argument with associated inputs based on the GraphQL type definitions that expose filtering criteria. You can see the full list of filtering criteria in the documentation at https://grandstack.io/docs/graphql-filtering.html.

4.7.1 Filter argument

In the following listing we use the filter argument to search for business names that contain "Brew".

Listing 4.18 GraphQL query

```
{
  Business(filter: { name_contains: "Brew" }) {
    name
    address
  }
}
```

Our results now show businesses that match the filtering criteria and only businesses that contain the string "Brew" in their name are returned.


```
{
  "data": {
    "Business": [
      {
        "name": "KettleHouse Brewing Co.",
        "address": "313 N 1st St W"
      },
      {
        "name": "Imagine Nation Brewing",
        "address": "1151 W Broadway St"
      },
```

```
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St"
      }
    ]
  }
}
```

4.7.2 Nested filter

To filter based on the results of nested fields applied to the root, we can nest our filter arguments. In the following listing we search for businesses where the name contains "Brew" and have at least one review with at least a 4.75 rating.

Listing 4.19 GraphQL query

```
{
  Business(
    filter: { name_contains: "Brew", reviews_some: { stars_gte: 4.75 } }
  ) {
    name
    address
  }
}
```

If we inspect the results of this GraphQL query we can see two matching businesses.

```
{
  "data": {
    "Business": [
      {
        "name": "KettleHouse Brewing Co.",
        "address": "313 N 1st St W"
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St"
      }
    ]
  }
}
```

4.7.3 Logical operators: AND, OR

Filters can be wrapped in logical operators `OR` and `AND`. For example, we can search for businesses in either the Coffee or Breakfast category by using an `OR` operator in the filter argument, as shown in the following listing.

Listing 4.20 GraphQL query

```
{
  Business(
    filter: {
      OR: [
        { categories_some: { name: "Coffee" } }
```

```
        { categories_some: { name: "Breakfast" } }
      ]
    }
  ) {
    name
    address
    categories {
      name
    }
  }
}
```

This GraphQL query yields businesses that are connected to either the Coffee or Breakfast category.

```
{
  "data": {
    "Business": [
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St",
        "categories": [
          {
            "name": "Restaurant"
          },
          {
            "name": "Breakfast"
          }
        ]
      },
      {
        "name": "Market on Front",
        "address": "201 E Front St",
        "categories": [
          {
            "name": "Coffee"
          },
          {
            "name": "Restaurant"
          },
          {
            "name": "Cafe"
          },
          {
            "name": "Deli"
          },
          {
            "name": "Breakfast"
          }
        ]
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St",
        "categories": [
          {
```

```
            "name": "Coffee"
          }
        ]
      }
    ]
  }
}
```

4.7.4 Filtering in selections

Filters can also be used throughout the selection set to apply the filter at the level of the selection. For example, let's say we want to find all Coffee or Breakfast businesses, but only view reviews containing the phrase "breakfast sandwich". See the following listing.

Listing 4.21 GraphQL query

```
{
  Business(
    filter: {
      OR: [
        { categories_some: { name: "Coffee" } }
        { categories_some: { name: "Breakfast" } }
      ]
    }
  ) {
    name
    address
    reviews(filter: { text_contains: "breakfast sandwich" }) {
      stars
      text
    }
  }
}
```

Because the filter was applied at the `reviews` selection, businesses that don't have any reviews containing the phrase "breakfast sandwich" are still shown in the results; however, only reviews containing that phrase are shown.

```
{
  "data": {
    "Business": [
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St",
        "reviews": [
          {
            "stars": 4,
            "text": "Best breakfast sandwich at the Farmer's Market. Always
     get the works."
          }
        ]
      },
      {
```

```
        "name": "Market on Front",
        "address": "201 E Front St",
        "reviews": []
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St",
        "reviews": []
      }
    ]
  }
}
```

4.8 Working with temporal fields

Neo4j supports native temporal types as properties on nodes and relationships. These types include Date, DateTime, and LocalDateTime. With neo4j-graphql.js you can use these temporal types in your GraphQL schema.

4.8.1 Using temporal fields in queries

Temporal types expose their date components (such as day, month, year, hour, etc.) as fields, as well as a formatted field which is the ISO-8601 string representation of the temporal value. The specific fields available vary depending on which temporal is used. See the following listing.

Listing 4.22 GraphQL query

```
{
  Review(first: 3, orderBy: date_desc) {
    stars
    date {
      formatted
    }
    business {
      name
    }
  }
}
```

Because we specified the `formatted` field on our date property, we see that in the results.

```
{
  "data": {
    "Review": [
      {
        "stars": 3,
        "date": {
          "formatted": "2018-09-10"
        },
        "business": {
          "name": "Imagine Nation Brewing"
        }
      },
```

```
      {
        "stars": 5,
        "date": {
          "formatted": "2018-08-11"
        },
        "business": {
          "name": "Zootown Brew"
        }
      },
      {
        "stars": 4,
        "date": {
          "formatted": "2018-03-24"
        },
        "business": {
          "name": "Market on Front"
        }
      }
    ]
  }
}
```

4.8.2 DateTime filters

Temporal fields are also included in the generated filtering enums, allowing for filtering using dates and date ranges. In the following listing we search for reviews created before January 1, 2017.

Listing 4.23 GraphQL query

```
{
  Review(
    first: 3
    orderBy: date_desc
    filter: { date_lte: { year: 2017, month: 1, day: 1 } }
  ) {
    stars
    date {
      formatted
    }
    business {
      name
    }
  }
}
```

We can see the results are now ordered by the `date` field.

```
{
  "data": {
    "Review": [
      {
        "stars": 5,
        "date": {
          "formatted": "2016-11-21"
        },
```

```
        "business": {
          "name": "Hanabi"
        }
      },
      {
        "stars": 5,
        "date": {
          "formatted": "2016-07-14"
        },
        "business": {
          "name": "KettleHouse Brewing Co."
        }
      },
      {
        "stars": 4,
        "date": {
          "formatted": "2016-01-03"
        },
        "business": {
          "name": "KettleHouse Brewing Co."
        }
      }
    ]
  }
}
```

4.9 Working with spatial data

Neo4j currently supports the spatial Point type, which can represent both 2D (such as latitude and longitude) and 3D (such as x,y,z or latitude, longitude, height) points. neo4j-graphql.js makes available the Point type for use in your GraphQL type definitions. The GraphQL schema augmentation process will translate the location field to a `_Neo4jPoint` type in the augmented schema.

4.9.1 The point type in selections

`Point` type fields are object fields in the GraphQL schema, so let's retrieve the latitude and longitude fields for our matching businesses by adding those fields to our selection set as shown in the following listing.

Listing 4.24 GraphQL query

```
{
  Business(first: 3, orderBy: name_asc) {
    name
    location {
      latitude
      longitude
    }
  }
}
```

Now, in the GraphQL query result we see longitude and latitude included for each business.

```
{
  "data": {
    "Business": [
      {
        "name": "Ducky's Car Wash",
        "location": {
          "latitude": 37.575968,
          "longitude": -122.336041
        }
      },
      {
        "name": "Hanabi",
        "location": {
          "latitude": 37.582598,
          "longitude": -122.351519
        }
      },
      {
        "name": "Imagine Nation Brewing",
        "location": {
          "latitude": 46.876672,
          "longitude": -114.009628
        }
      }
    ]
  }
}
```

4.9.2 Distance filter

When querying using point data, often we want to find things that are close to other things. For example, what businesses are within 1.5km of me? We can accomplish this using the auto-generated filter argument in the following listing.

Listing 4.25 GraphQL query

```
{
  Business(
    filter: {
      location_distance_lt: {
        point: { latitude: 37.563675, longitude: -122.322243 }
        distance: 3500
      }
    }
  ) {
    name
    address
    city
    state
  }
}
```

For points using the Geographic coordinate reference system (latitude and longitude), distance is measured in meters.

```
{
  "data": {
    "Business": [
      {
        "name": "Hanabi",
        "address": "723 California Dr",
        "city": "Burlingame",
        "state": "CA"
      },
      {
        "name": "Ducky's Car Wash",
        "address": "716 N San Mateo Dr",
        "city": "San Mateo",
        "state": "CA"
      },
      {
        "name": "Neo4j",
        "address": "111 E 5th Ave",
        "city": "San Mateo",
        "state": "CA"
      }
    ]
  }
}
```

4.10 Adding custom logic

We've seen basic querying operations created by neo4j-graphql.js. Often, we want to add custom logic to our API. For example, we may want to calculate the most popular business or recommend businesses to users. There are two options for adding custom logic to your API using neo4j-graphql.js: 1) the `@cypher` schema directive, and 2) by implementing custom resolvers.

4.10.1 The @cypher directive

We expose Cypher through GraphQL via the `@cypher` directive. Annotate a field in your schema with the `@cypher` directive to map the results of that query to the annotated GraphQL field. The `@cypher` directive takes a single argument `statement` which is a Cypher statement. Parameters are passed into this query at runtime, including `this` which is the currently resolved node as well as any field-level arguments defined in the GraphQL type definition.

> **NOTE** The `@cypher` directive feature requires the use of the APOC standard library plugin. Be sure you follow the steps to install APOC in the Project Setup section of this chapter.

COMPUTED SCALAR FIELDS

We can use the `@cypher` directive to define a custom scalar field, defining a computed field in our schema. Here we add an `averageStars` field to the `Business` type, which

calculates the average stars of all reviews for the business using this variable, as shown in the following listing.

Listing 4.26 index.js: GraphQL type definitions

```
type Business {
    businessId: ID!
    averageStars: Float! @cypher(statement:"MATCH (this)<-[:REVIEWS]-
     (r:Review) RETURN avg(r.stars)")
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
}
```

We need to restart our GraphQL server because we have modified the type definitions:

```
node index.js
```

Now, let's include the `averageStars` field in our GraphQL query, as shown in the following listing.

Listing 4.27 GraphQL query including averageStars field

```
{
  Business {
    name
    averageStars
  }
}
```

And we see in the results that the computed value for `averageStars` is now included.

```
{
  "data": {
    "Business": [
      {
        "name": "Hanabi",
        "averageStars": 5
      },
      {
        "name": "Zootown Brew",
        "averageStars": 5
      },
      {
        "name": "Ninja Mike's",
        "averageStars": 4.5
      }
    ]
  }
}
```

The generated Cypher query includes the annotated Cypher query as a sub-query, preserving the single database call to resolve the GraphQL request.

COMPUTED OBJECT AND ARRAY FIELDS

We can also use the `@cypher` schema directive to resolve object and array fields. Let's add a recommended business field to the `Business` type. We'll use a simple Cypher query to find common businesses that other users reviewed. For example, if a user likes "Market on Front", we could recommend other businesses that users who reviewed "Market on Front" also reviewed, as shown in the following listing.

Listing 4.28 Cypher

```
MATCH (b:Business {name: "Market on Front"})<-[:REVIEWS]-(:Review)<-[:WROTE]-
     (:User)-[:WROTE]->(:Review)-[:REVIEWS]->(rec:Business)
WITH rec, COUNT(*) AS score
RETURN rec ORDER BY score DESC
```

We can use this Cypher query in our GraphQL schema by including it in a `@cypher` directive on the `recommended` field in our `Business` type definition, as shown in the following listing.

Listing 4.29 index.js: GraphQL type definitions

```
type Business {
    businessId: ID!
    averageStars: Float! @cypher(statement:"MATCH (this)<-[:REVIEWS]-
     (r:Review) RETURN avg(r.stars)")
    recommended(first: Int = 1): [Business] @cypher(statement: """
      MATCH (this)<-[:REVIEWS]-(:Review)<-[:WROTE]-(:User)-[:WROTE]-
     >(:Review)-[:REVIEWS]->(rec:Business)
      WITH rec, COUNT(*) AS score
      RETURN rec ORDER BY score DESC LIMIT $first
    """)
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
}
```

We also define a `first` field argument, which is passed to the Cypher query included in the `@cypher` directive and acts as a limit on the number of recommended businesses returned.

CUSTOM TOP-LEVEL QUERY FIELDS

Another helpful way to use the `@cypher` directive is as a custom query or mutation field. For example, let's see how we can add full-text query support to search for businesses. Applications often use full-text search to correct for things such as misspellings in user input using fuzzy matching.

In Neo4j, we can use full-text search by first creating a full-text index, as shown in the following listing.

Listing 4.30 Cypher: create full-text index

```
CALL db.index.fulltext.createNodeIndex("businessNameIndex",
     ["Business"],["name"])
```

Then to query the index, in this case we misspell "coffee" but including the ~ character enables fuzzy matching, ensuring we still find what we're looking for, as shown in the following listing.

Listing 4.31 Cypher: querying the full-text index

```
CALL db.index.fulltext.queryNodes("businessNameIndex", "cofee~")
```

Wouldn't it be nice to include this fuzzy matching full-text search in our GraphQL API? To do that let's create a Query field called `fuzzyBusinessByName` that takes a search string and searches for businesses, as shown in the following listing.

Listing 4.32 index.js: GraphQL type definitions

```
type Query {
    fuzzyBusinessByName(searchString: String): [Business] @cypher(
      statement: """
        CALL db.index.fulltext.queryNodes( 'businessNameIndex',
     $searchString+'~')
        YIELD node RETURN node
      """
   )
}
```

Again, since we've updated the type definitions, we must restart the GraphQL API application:

```
node index.js
```

If we check the Docs tab in GraphQL Playground, we'll see a new Query field `fuzzyBusinessByName`, and we can now search for business names using this fuzzy matching (see the following listing).

Listing 4.33 GraphQL query

```
{
  fuzzyBusinessByName(searchString: "libary") {
    name
  }
}
```

Because we're using full-text search, even though we spell "library" incorrectly, we still find matching results.

```
{
  "data": {
    "fuzzyBusinessByName": [
      {
        "name": "Missoula Public Library"
```

```
      }
    ]
  }
}
```

The `@cypher` schema directive is a powerful way to add custom logic and advanced functionality to our GraphQL API. We can also use the `@cypher` directive for authorization features, accessing values such as authorization tokens from the request object, a pattern that will be discussed in a later chapter when we explore different options for adding authorization to our API.

4.10.2 Implementing custom resolvers

While the `@cypher` directive is one way to add custom logic, in some cases we may need to implement custom resolvers that implement logic not able to be expressed in Cypher. For example, we may need to fetch data from another system, or apply custom validation rules. In these cases, we can implement a custom resolver and attach it to the GraphQL schema so that resolver is called to resolve our custom field instead of relying on the generated Cypher query by neo4j-graphql.js to resolve the field.

In our example, let's imagine there is an external system that can be used to determine current wait times at businesses. We want to add an additional `waitTime` field to the `Business` type in our schema and implement the resolver logic for this field to use this external system.

To do this, we first add the field to our schema, adding the `@neo4j_ignore` directive to ensure the field is excluded from the generated Cypher query. This is our way of telling neo4j-graphql.js that a custom resolver will be responsible for resolving this field and we don't expect it to be fetched from the database automatically, as shown in the following listing.

Listing 4.34 index.js: GraphQL type definitions

```
type Business {
    businessId: ID!
    waitTime: Int! @neo4j_ignore
    averageStars: Float!
      @cypher(
        statement: "MATCH (this)<-[:REVIEWS]-(r:Review) RETURN avg(r.stars)"
      )
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review] @relation(name: "REVIEWS", direction: "IN")
    categories: [Category] @relation(name: "IN_CATEGORY", direction: "OUT")
  }
```

Next, we create a resolver map with our custom resolver. We didn't have to create this previously because neo4j-graphql.js generated our resolvers for us. Our wait time calculation will be selecting a value at random, but we could implement any custom logic

here to determine the waitTime value, such as making a request to a third-party API, as shown in the following listing.

Listing 4.35 index.js: creating a resolver map

```
const resolvers = {
  Business: {
    waitTime: (obj, args, context, info) => {
      const options = [0, 5, 10, 15, 30, 45];
      return options[Math.floor(Math.random() * options.length)];
    }
  }
};
```

Then we add this resolver map to the parameters passed to makeAugmentedSchema, as shown in the following listing.

Listing 4.36 index.js: generating the GraphQL schema

```
const schema = makeAugmentedSchema({
  typeDefs,
  resolvers
});
```

Now we restart the GraphQL API application because we've updated the code:

```
node index.js
```

After restarting, in GraphQL Playground if we check the Docs for the Business type, we'll see our new field waitTime on the Business type.

Now, let's search for restaurants and see what their wait times are by including the waitTime field in the selection set in the following listing.

Listing 4.37 GraphQL query

```
{
  Business(filter: { categories_some: { name: "Restaurant" } }) {
    name
    waitTime
  }
}
```

In the results we now see a value for the wait time. Your results will of course vary because the value is randomized.

```
{
  "data": {
    "Business": [
      {
        "name": "Ninja Mike's",
        "waitTime": 5
      },
      {
        "name": "Market on Front",
        "waitTime": 45
      },
```

```
      {
        "name": "Hanabi",
        "waitTime": 45
      }
    ]
  }
}
```

4.11 Inferring GraphQL schema from an existing database

Typically, when we start a new application, we don't have an existing database and follow the GraphQL-First development paradigm by starting with type definitions. However, in certain cases we may have an existing Neo4j database populated with data. In those cases, it can be convenient to generate GraphQL type definitions based on the existing database that can then be fed into `makeAugmentedSchema` to generate a GraphQL API for the existing database. We can do this with the use of the `inferSchema` functionality in neo4j-graphql.js.

This Node.js script will connect to our Neo4j database and infer the GraphQL type definitions that describe this data, then write those type definitions to a file named schema.graphql, as shown in the following listing.

Listing 4.38 infer.js: inferring GraphQL type definitions

```
const neo4j = require("neo4j-driver");
const { inferSchema } = require("neo4j-graphql-js");
const fs = require("fs");

const driver = neo4j.driver(
  "bolt://localhost:7687",
  neo4j.auth.basic("neo4j", "letmein")
);

const schemaInferenceOptions = {
  alwaysIncludeRelationships: false
};

inferSchema(driver, schemaInferenceOptions).then(result => {
  fs.writeFile("schema.graphql", result.typeDefs, err => {
    if (err) throw err;
    console.log("Updated schema.graphql");
    process.exit(0);
  });
});
```

Then we can load this schema.graphql file in the following listing and pass the type definitions into `makeAugmentedSchema`.

Listing 4.39 Initial GraphQL API code

```
// Load GraphQL type definitions from schema.graphql file
const typeDefs =
     fs.readFileSync(path.join(__dirname,"schema.graphql")).toString("utf-8");
```

Up to now, all of our GraphQL querying has been done using GraphQL Playground, which is great for testing and development, but typically our goal is to build an application that queries the GraphQL API. In the next few chapters, we'll start to build out the user interface for our business reviews application using React and Apollo Client. Along the way, we'll learn more about GraphQL concepts such as mutations, fragments, interface types, and more!

Exercises

1. Query the GraphQL API we created in this chapter using GraphQL Playground to find:
   - Which users have reviewed the business named "Hanabi"?
   - Find any reviews that contain the word "comfortable". What business(es) are they reviewing?
   - Which users have given no five-star reviews?
2. Add a `@cypher` directive field to the `Category` type that computes the number of businesses in each category. How many businesses are in the "Coffee" category?
3. Create a Neo4j Sandbox instance at https://sandbox.neo4j.com choosing from any of the pre-populated datasets. Using the `inferSchema` method from neo4j-graphql.js, create a GraphQL API for this Neo4j Sandbox instance without manually writing GraphQL type definitions. What data can you query for using GraphQL?

Refer to the book's GitHub repository to see the exercise solutions: https://github.com/johnymontana/fullstack-graphql-book.

Summary

- Common problems that arise when building GraphQL APIs include the n+1 query problem, schema duplication, and a large amount of boilerplate data-fetching code.
- GraphQL database integrations like neo4j-graphql.js can help mitigate these problems by generating database queries from GraphQL requests, driving database schema from GraphQL type definitions, and auto-generating a GraphQL API from GraphQL type definitions.
- neo4j-graphql.js makes it easy to build GraphQL APIs backed by a Neo4j database by generating resolvers for data fetching and adding filtering, ordering, and pagination to the generated API.
- Custom logic can be added by using the `@cypher` schema directive to define custom logic for fields, or by implementing custom resolvers and attaching them to the GraphQL schema.
- If we have an existing Neo4j database, we can use the `inferSchema` functionality of neo4j-graphql.js to generate GraphQL type definitions and a GraphQL API on top of the existing database.

index

Symbols

A

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

W