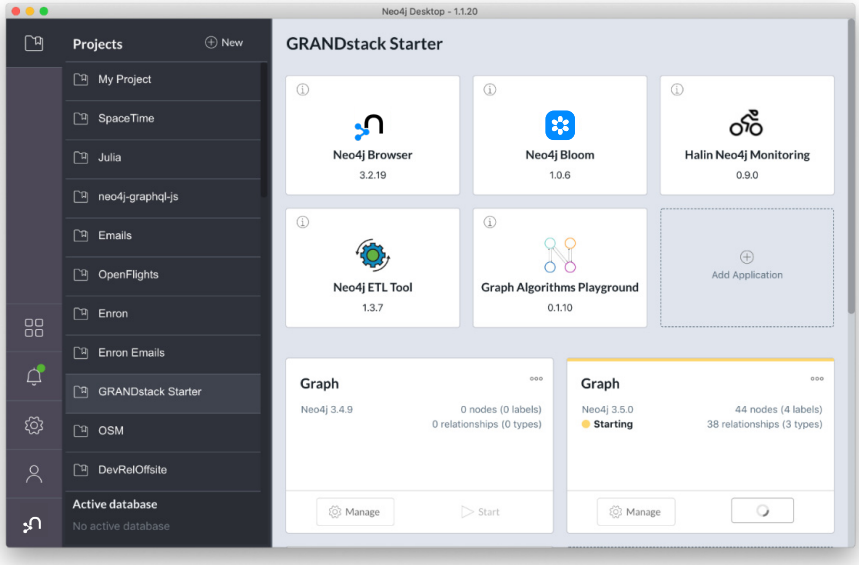

Figure 1.9 Neo4j Desktop.

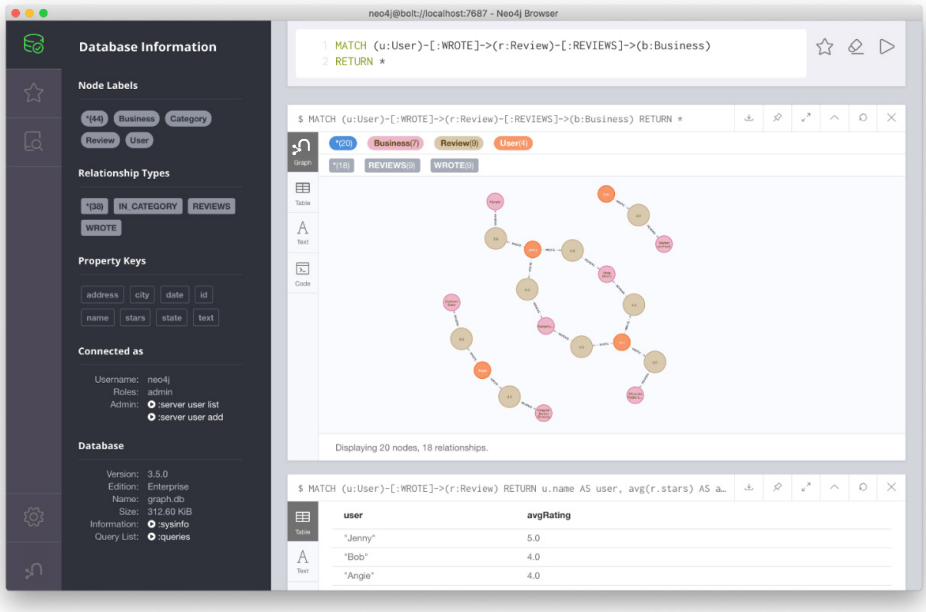

Figure 1.10 Neo4j Browser.

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();
)
```

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

Importing the module.

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".

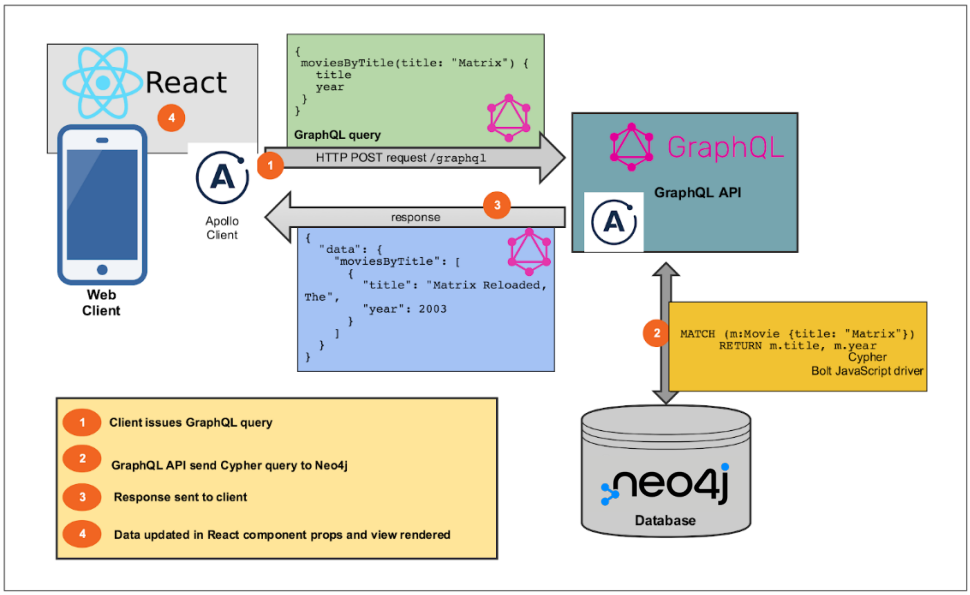

Figure 1.11 Following a movie search request through a GRANDstack application.

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.

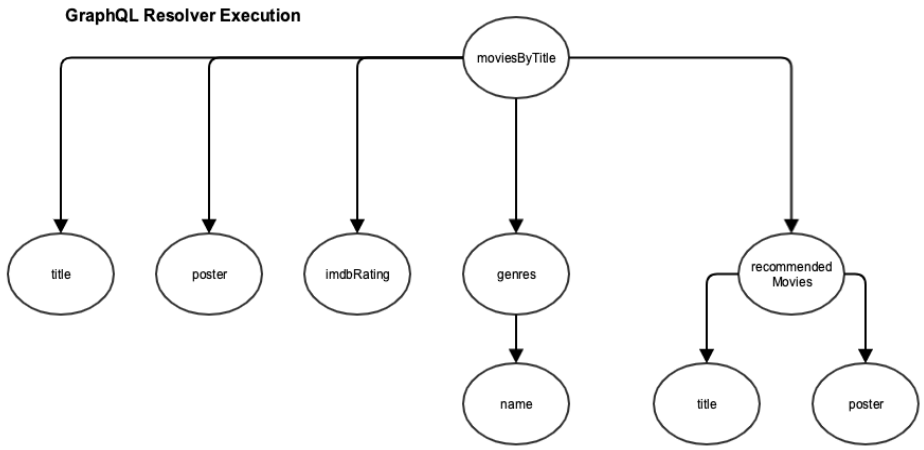

Figure 1.12 **GraphQL resolver functions are called in a nested fashion.**

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.

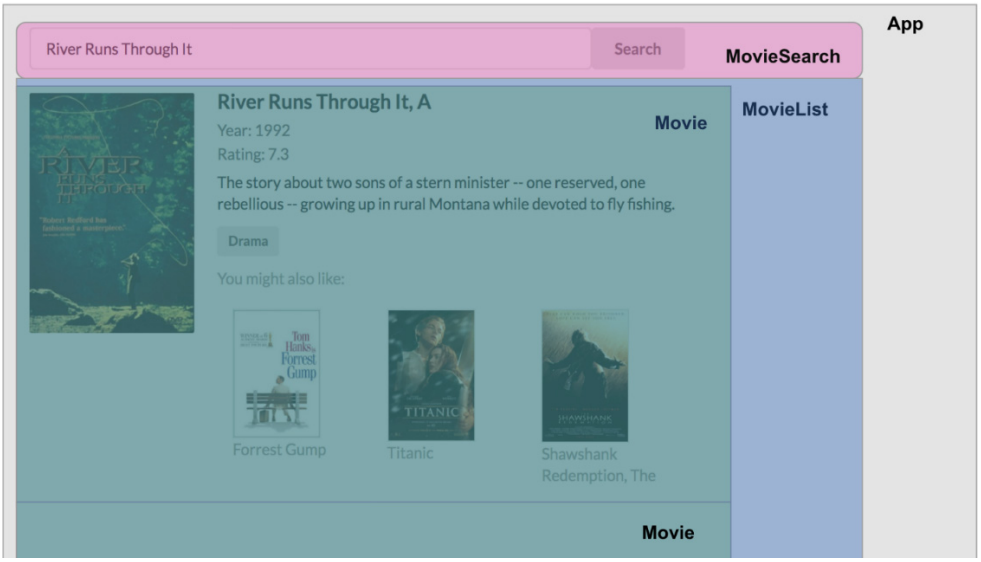

Figure 1.13 **React components are composed together to build a complex user interface.**

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.