Оглавление

Вступление

Благодарим вас за выбор «Разработка веб-приложения GraphQL с React, Node.js и Neo4j». Цель этой книги – показать, как можно использовать GraphQL, React, Apollo и базу данных Neo4j (так называемый стек GRAND) для создания сложных приложений, интенсивно применяющих данные. Возможно, вам интересно, почему мы выбрали именно эту комбинацию технологий. Я надеюсь, что в процессе чтения вы оцените продуктивность, производительность и интуитивно понятные преимущества использования графовой модели данных на всем протяжении – от базы данных до API и в коде, выбирающем данные на стороне клиента.

Я мечтал найти такую книгу, когда, будучи молодым инженером, получил работу в небольшом стартапе, занимающемся созданием полнофункционального веб-приложения. Мы потратили месяцы на оценку стека технологий и изучение способов их комбинирования друг с другом. В конце концов мы приступили к работе с применением технологий, которые нас устраивали, но на их выбор потребовалось много итераций.

GraphQL – это технология, коренным образом изменившая подходы к разработке веб-приложений. Эта книга посвящена GraphQL; однако одного лишь понимания, как создавать серверы и писать операции GraphQL, недостаточно для реализации приложений полного цикла. Нужно также подумать о том, как организовать выборку данных из GraphQL и управление состоянием внешнего приложения, как защитить API, как развернуть приложение, и учесть массу других соображений. Вот почему эта книга не только о GraphQL; она рассказывает об использовании GraphQL в целом, показывая, как разные части сочетаются друг с другом. Если перед вами стоит задача создать приложение полного цикла с использованием GraphQL, то эта книга для вас!

О книге

Цель книги «Разработка веб-приложения GraphQL с React, Node.js и Neo4j» – показать, как разные части стека GraphQL сочетаются друг с другом в полномасштабных приложениях и как разработчики могут использовать онлайн-службы для поддержки разработки и развертывания. С этой целью в каждой главе будут представляться новые идеи и понятия и применяться для создания и развертывания полномасштабного приложения.

Кому адресована эта книга

Эта книга предназначена для веб-разработчиков полного цикла, заинтересованных в технологии GraphQL и имеющих хотя бы базовый уровень понимания Node.js API и особенностей клиентских приложений на JavaScript, использующих этот API. Прочитав эту книгу, читатель получит базовое представление о Node.js и клиентском JavaScript, но, что особенно важно, приобретет мотивацию для освоения приемов создания служб и приложений с использованием GraphQL.

Организация книги

Эта книга состоит из девяти глав, разделенных на три части. В каждой главе обсуждаются новые концепции и технологии в контексте создания полномасштабных приложений.

В первой части вы познакомитесь с GraphQL – графовой базой данных для Neo4j – и собственно с понятием графов:

- в главе 1 обсуждаются компоненты полномасштабных приложений GraphQL, включая введение во все конкретные технологии, используемые в этой книге (GraphQL, React, Apollo и база данных Neo4j);
- глава 2 знакомит с GraphQL и основами создания GraphQL API (определения типов и функции распознавания);
- глава 3 знакомит с графовой базой данных Neo4j, моделью графа свойств и языком запросов Cypher;
- глава 4 демонстрирует возможности GraphQL при работе с графовой базой данных Neo4j посредством библиотеки Neo4j GraphQL.

Во второй части мы сосредоточимся на разработке клиентского приложения с использованием React:

- глава 5 знакомит с инфраструктурой библиотеки React и концепциями ее применения, которые пригодятся, когда мы приступим к созданию примера клиентского приложения;
- глава 6 показывает, как организовать выборку данных и управление состоянием клиента с помощью React и GraphQL при работе с GraphQL API, созданным в предыдущих главах.

В третьей части мы займемся защитой приложения и его развертыванием с использованием облачных служб:

- глава 7 показывает, как защитить приложение, используя GraphQL и Auth0;
- глава 8 знакомит с облачными службами, обычно используемыми для развертывания баз данных, GraphQL API и приложений React;
- глава 9 завершает книгу обсуждением абстрактных типов в GraphQL, разбиением наборов данных на страницы с помощью курсоров и обработки свойств отношений в GraphQL.

Эту книгу следует читать от начала до конца, потому что каждая следующая глава основывается на предыдущих, и все они описывают процесс создания полномасштабного приложения. Читатели могут сосредоточиться на отдельных главах, погрузившись в интересующие их темы, но при этом желательно прочитать предыдущие главы, чтобы узнать, как и почему созданы те или иные части приложения.

О примерах программного кода

Эта книга также содержит множество примеров программного кода как в пронумерованных листингах, так и в виде включений в обычный текст. В обоих случаях исходный код оформлен `шрифтом фиксированной ширины`, чтобы вам было проще отличать его от основного текста. Иногда вам будет встречаться код, оформленный **`жирным моноширинным шрифтом`**, чтобы выделить изменившиеся фрагменты, по сравнению с предыдущими шагами, например когда в существующую строку кода добавляется что-то новое.

Во многих случаях исходный код переформатирован, чтобы уместить его по ширине книжной страницы. В частности, мы добавили разрывы строк и отступы. В редких случаях даже этого было недостаточно, и мы добавили маркеры продолжения строки (➥). Кроме того, мы удалили комментарии из листингов, которые подробно описываются в тексте. Многие листинги сопровождаются дополнительными примечаниями, описывающими важные понятия.

Получить выполняемые фрагменты кода можно из электронной версии книги по адресу **https://livebook.manning.com/book/fullstack-graphql-applications**. Все примеры, что приводятся в книге, доступны для загрузки на веб-сайте издательства Manning (**www.manning.com**) и в репозитории GitHub по адресу **https://github.com/johnymontana/fullstack-graphql-book**.

Требования к программному обеспечению

Для следования за примерами в книге необходимо установить последнюю версию Node.js. Все примеры я опробовал с версией v16. По своему опыту рекомендую использовать инструмент `nvm` для установки и управления версиями Node.js. Инструкции по установке и использованию `nvm` можно найти по адресу **https://github.com/nvm-sh/nvm**.

Мы также будем применять несколько (бесплатных) онлайн-служб для развертывания. Доступ к большинству из них можно получить с помощью учетной за-

писи GitHub, поэтому обязательно зарегистрируйтесь на GitHub (**https://github.com/**), если вы этого еще не сделали.

Живое обсуждение книги

Приобретая книгу «Разработка веб-приложения GraphQL с React, Node.js и Neo4j», вы получаете бесплатный доступ к онлайн-платформе liveBook для чтения книг издательства Manning. Благодаря эксклюзивным возможностям этой платформы вы можете оставлять свои комментарии к книге как в целом, так и к определенным разделам или абзацам, добавлять заметки для себя, задавать технические вопросы и отвечать на них, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу **https://livebook.manning.com/book/fullstack-graphql-applications/discussion**. Узнать больше о форумах Manning и познакомиться с правилами поведения можно по адресу **https://livebook.manning.com/discussion**.

Издательство Manning обязуется предоставить своим читателям место встречи, где может состояться содержательный диалог между отдельными читателями и между читателями и автором. Но со стороны авторов отсутствуют какие-либо обязательства уделять форуму какое-то определенное внимание – их присутствие на форуме остается добровольным (и неоплачиваемым). Мы предлагаем задавать авторам стимулирующие вопросы, чтобы их интерес не угасал! Форум и архив с предыдущими обсуждениями остаются доступны на сайте издательства, пока книга продолжает издаваться.

Другие онлайн-ресурсы

Вам обязательно пригодится документация к библиотеке Neo4j GraphQL, доступная по адресу **https://neo4j.com/docs/graphql-manual/current/**. В числе других полезных онлайн-ресурсов можно назвать бесплатные онлайн-курсы на GraphAcademy (**https://graphacademy.neo4j.com/**), сайт сообщества Neo4j (**https://community.neo4j.com/**).

Об авторе

Уильям Лион (William Lyon) – консультант в Neo4j, где он помогает разработчикам успешно создавать приложения с графовыми базами данных. До прихода в Neo4j работал инженером-программистом в стартапах, занимающихся созданием финансовых систем, мобильных приложений для индустрии недвижимости и прогнозными API. Имеет степень магистра информатики, полученную в университете штата Монтана, и ведет блог на **lyonwj.com**.

Об иллюстрации на обложке

Рисунок на обложке книги называется «Dame de l'Isle de Tinne» (леди с острова Тинне) из коллекции Жака Грассе де Сен-Совер (Jacques Grasset de Saint-Sauveur), опубликованной в 1797 году. Все иллюстрации в этой коллекции тщательно прорисованы и раскрашены вручную.

В те дни по одежде было легко определить, где живет человек, чем занимается и какое положение занимает в обществе. Мы в издательстве Manning славим изобретательность, предприимчивость и радость компьютерного бизнеса обложками книг, изображающими богатство региональных различий многовековой давности, оживших благодаря иллюстрациям, таким как эта.

Часть I

Введение в стек GraphQL

Прежде чем начать путешествие в стек GraphQL, рассмотрим технологии, которые будут использоваться, и мощную концепцию графового мышления. Этот раздел посвящен серверной части нашего приложения и, в частности, базе данных и GraphQL API.

В главе 1 вы познакомитесь с компонентами приложения GraphQL полного цикла и с конкретными технологиями, которые будут использоваться на протяжении всей книги: GraphQL, React, Apollo и БД Neo4j. В главе 2 мы с головой погрузимся в GraphQL и основы создания GraphQL API. В главе 3 исследуем графовую базу данных Neo4j, модель данных графа свойств и язык запросов Cypher. В главе 4 посмотрим, как использовать интеграцию базы данных для поддержки GraphQL и, в частности, библиотеку Neo4j GraphQL для создания GraphQL API, поддерживаемых графовой базой данных. По завершении первой части книги у вас будет готовая к экспериментам база данных и начальное приложение GraphQL API, после чего вы сможете перейти ко второй части книги и приступить к созданию внешнего интерфейса.

Глава 1

Что такое стек GraphQL?

В этой главе:

- компоненты, составляющие типичное приложение GraphQL полного цикла;
- технологии, используемые в книге (GraphQL, React, Apollo и БД Neo4j), и их сочетание в контексте приложения полного цикла;
- требования к приложению, которое будет создаваться на протяжении всей книги.

1.1. Обзор стека GraphQL

В этой главе мы познакомимся с технологиями, которые будут использоваться на протяжении всей книги:

- GraphQL – для создания API;
- React – для создания пользовательского интерфейса и клиентского веб-приложения на JavaScript;
- Apollo – инструменты для работы с GraphQL как на сервере, так и на клиенте;
- Neo4j – база данных, которую мы используем для хранения данных приложения и управления ими.

Создание приложения GraphQL полного цикла предполагает работу с многоуровневой архитектурой, широко известной как *трехуровневое приложение*, которая состоит из внешнего интерфейса, уровня API и базы данных. На рис. 1.1 можно видеть отдельные компоненты приложения GraphQL полного цикла и их взаимодействие друг с другом.

На протяжении всей книги мы будем использовать эти технологии и компоненты для создания простого приложения, подробно обсуждая каждый в процессе реализации. А основные требования к приложению будут перечислены в последнем разделе этой главы.

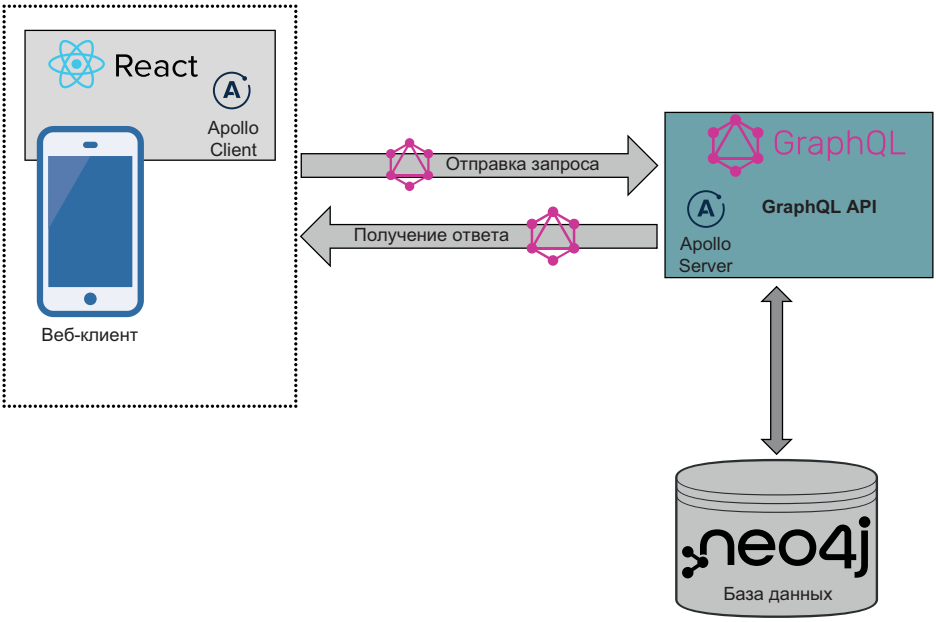

Рис. 1.1. Компоненты приложения GraphQL полного цикла: GraphQL, React, Apollo и база данных Neo4j

Основное внимание в данной книге уделяется изучению приемов создания приложений GraphQL, поэтому рассматривать GraphQL мы будем на примере приложения полного цикла в комплексе с другими технологиями, включая разработку схемы, интеграцию с базой данных, создание веб-интерфейса, обращающегося к GraphQL API, добавление аутентификации и т. д. Учитывая все это, книга предполагает наличие у читателя некоторых базовых знаний об особенностях создания веб-приложений, но не требует опыта работы с каждой конкретной технологией. Чтобы добиться успеха, читатель должен иметь базовые навыки программирования на JavaScript как на стороне клиента, так и в Node.js, а также владеть такими понятиями, как API (Application Programming Interface – прикладной программный интерфейс) и базы данных. Для опробования примеров должен быть установлен пакет `node` и желательно уметь пользоваться инструментом командной строки `npm` (или `yarn`) для создания проектов Node.js и установки зависимостей. Мы будем использовать последнюю LTS-версию Node.js (на момент написания этих строк – версия 16.14.2), которую можно получить по адресу **https://nodejs.org/**. Для управления версиями Node.js можно использовать диспетчер версий `nvm`. Дополнительную информацию вы найдете по адресу **https://github.com/nvm-sh/nvm**.

Перед обсуждением каждой технологии дается краткое введение и по мере необходимости предлагаются дополнительные источники более подробной информации. Также в процессе обсуждения конкретных технологий, используемых вместе с GraphQL, будут перечисляться другие аналогичные технологии (технология создания веб-интерфейса Vue, которую можно использовать вместо React). В конечном счете цель этой книги – показать, как эти технологии сочетаются друг с другом, и помочь читателю составить полную картину стека технологий для создания приложений на основе GraphQL.

1.2. GraphQL

GraphQL – это спецификация для создания API. Она описывает язык запросов к API и способ выполнения этих запросов. При создании GraphQL API разработчик описывает доступные данные, используя строгую систему типов. Эти описания, также определяющие точки входа в API, становятся спецификацией, основываясь на которой, клиент может запросить необходимые ему данные.

GraphQL обычно рассматривают как альтернативу REST – парадигме разработки API, наверняка знакомой вам. Это верное суждение, но лишь в некоторых случаях, потому что GraphQL также может обертывать существующие REST API или другие источники данных. Это обусловлено независимостью GraphQL от хранилища данных, благодаря которой GraphQL можно использовать с любыми источниками данных.

> *GraphQL – это язык запросов для API и среда выполнения этих запросов. GraphQL предоставляет полное и понятное описание данных, доступных в API, дает клиентам возможность запрашивать именно то, что им нужно, и ничего больше, тем самым упрощая развитие API с течением времени и давая разработчику мощные инструменты.*
>
> – graphql.org (**https://graphql.org/**)

А теперь более конкретно рассмотрим некоторые аспекты GraphQL.

1.2.1. *Определения типов в GraphQL*

GraphQL API организован не вокруг конечных точек, соответствующих ресурсам (как в REST), а вокруг определений, описывающих типы данных, поля и связи между ними. Эти определения типов становятся схемой API, который обслуживается одной конечной точкой.

Поскольку службы GraphQL могут быть реализованы на любом языке, для описания типов GraphQL используется свой универсальный язык определения схем GraphQL Schema Definition Language (SDL). Рассмотрим пример на рис. 1.2 – простое приложение для работы с фильмотекой. Представьте, что вас наняли для создания веб-сайта, позволяющего пользователям выполнять поиск сведений о фильмах в каталоге по их названиям, именам актеров и описаниям, а также показывать похожие фильмы, которые могут быть интересны пользователям.

Начнем с создания нескольких простых определений типов GraphQL (листинг 1.1), определяющих предметную область приложения.

Листинг 1.1. Простые определения типов для GraphQL API фильмотеки

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

`Movie` – **это тип объекта GraphQL, содержащего одно или несколько полей**

`title` – **это поле типа** `String`

Поля могут ссылаться на другие типы, например, в данном случае на список объектов типа `Actor`

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

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

`ActorId` – **обязательное (или непустое), на что указывает символ** `!`**, поле типа** `Actor`

`Query` – **специальный тип в GraphQL, определяющий точки входа в API**

Поля также могут иметь аргументы; в этом случае поле `movieSearch` **принимает обязательный строковый аргумент** `searchString`

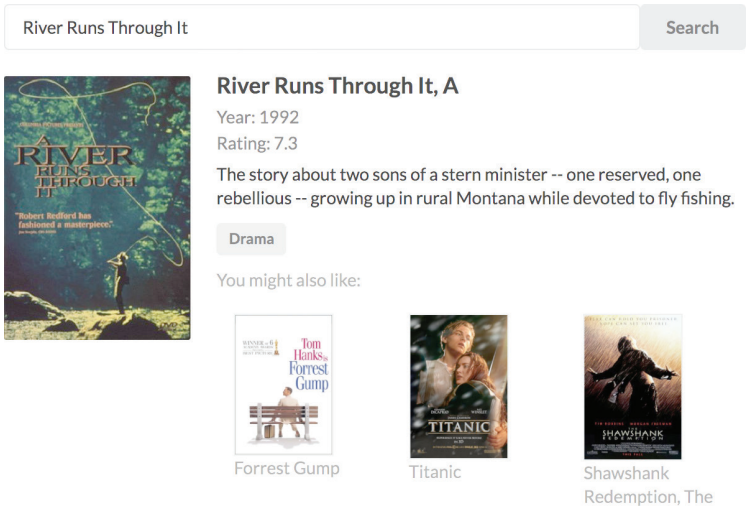

Рис. 1.2. Простое веб-приложение фильмотеки

Наши определения объявляют типы GraphQL, используемые в API, их поля и связи между ними. При определении типа объекта (например, `Movie`) также указываются все поля, доступные в объекте, и их типы (позже можно добавить дополнительные поля, используя ключевое слово `extend`). В этом примере поле `title` определяется со скалярным типом `String`, т. е. поле может содержать только одно значение. В отличие от него, поля объектных типов могут содержать несколько полей и ссылок на другие типы. В данном примере `actors` – это поле типа `[Actor]`, оно может содержать массив объектов `Actor` и определяет связь между типами `Movie` и `Actor` (такие связи образуют «граф»).

Поля могут быть необязательными или обязательными. Поле `actorId` в типе `Actor` является обязательным (т. е. оно не может быть пустым). Это означает, что каждый объект `Actor` должен иметь значение в поле `actorId`. Поля без восклицательного знака (`!`) в определении могут иметь значение `NULL`, т. е. они – необязательные.

Поля в типе `Query` определяют точки входа в службу GraphQL. Схемы GraphQL также могут содержать тип `Mutation`, определяющий точки входа для операций

записи в API. Третий особый тип, связанный с точками входа, – тип `Subscription`. Он определяет события, на которые клиент может подписаться.

> **ПРИМЕЧАНИЕ.** Здесь мы опускаем многие важные концепции GraphQL, такие как операции изменения, типы интерфейсов и объединений и т. д., но не волнуйтесь; мы только начинаем и скоро доберемся до них!

На этом этапе вам может быть интересно, где хранится граф GraphQL. Определяя типы GraphQL, мы фактически определяем граф. Граф – это структура данных, состоящая из узлов (сущностей или объектов) и отношений, соединяющих узлы. Именно такую структуру мы определили в описаниях типов на языке SDL. Определения выше задают простой граф со структурой, изображенной на рис. 1.3.

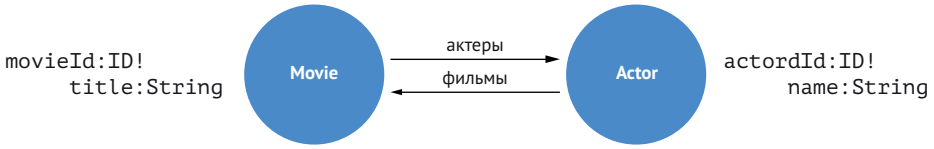

Рис. 1.3. Определения типов GraphQL для веб-приложения фильмотеки в виде графа

Графы предназначены для описания связанных данных, и здесь мы определили, как фильмы и актеры связаны между собой. GraphQL позволяет моделировать данные в виде графа и перемещаться по графу с помощью операций GraphQL.

Когда служба GraphQL получает запрос на выполнение операции, она проверяет и выполняет эту операцию в соответствии со схемой определений типов. Давайте рассмотрим пример запроса, который служба GraphQL может выполнить, руководствуясь заданными выше определениями типов.

1.2.2. *Запросы GraphQL*

Запросы GraphQL определяют порядок обхода графа данных в соответствии с определениями типов и запрашивают подмножество полей для возврата в ответе – это называется *выборкой множества*. Следующий запрос начинает обход графа с точки входа, заданной в поле запроса `allMovies`, и отыскивает актеров, связанных с каждым фильмом (листинг 1.2). Затем для каждого актера выполняется поиск других фильмов, в которых они снимались.

Листинг 1.2. Запрос GraphQL для поиска актеров и фильмов

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

Необязательное имя операции. По умолчанию используется имя query и его можно опустить. Имя операции – в данном случае `FetchSomeMovies` – тоже необязательное и может быть опущено

Здесь указывается точка входа, поле в типе `Query` или `Mutation`. В этом случае точкой входа для запроса является поле `allMovies` в типе `Query`

Выборка множества определяет поля, которые должны быть возвращены в ответ на запрос

Если предполагается вернуть поле объектного типа, следует определить вложенную выборку множества, описывающую возвращаемые поля

Для возврата полей объектов `Movie` необходимо определить вложенную выборку множества

```
  }
}
```

Обратите внимание, что наш запрос является вложенным и описывает порядок обхода графа связанных объектов (в данном случае фильмов и актеров). Этот обход и его результаты можно представить в виде графа данных визуально (рис. 1.4).

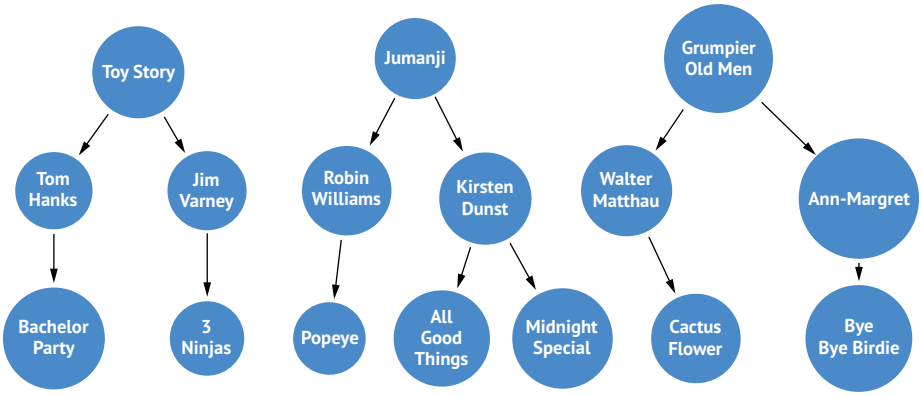

Рис. 1.4. Порядок обхода графа данных при выполнении запроса GraphQL

Обход графа данных можно представить визуально, но типичным результатом запроса GraphQL является документ JSON, как показано в листинге 1.3.

Листинг 1.3. Результаты запроса в формате JSON

```
"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"
            }
          ]
        }
      ]
    }
  ]
}
```

Как можно заметить, ответ соответствует форме полученной выборке множества – возвращаются только поля, указанные в запросе. Но откуда берутся данные? Логика получения данных GraphQL API определяется так называемыми *функциями разрешения* (resolver functions), которые определяют, какие данные соответ-

ствуют произвольному запросу GraphQL. GraphQL не зависит от типа хранилища, поэтому функции разрешения могут обращаться к одной или нескольким базам данных или извлекать данные из другого API – даже из REST API. Мы подробно рассмотрим эти функции в следующей главе.

1.2.3. *Преимущества GraphQL*

Теперь, увидев первый запрос GraphQL, вы можете подумать: «Все это хорошо и даже замечательно, но все те же данные я могу получать с помощью REST. В чем же тогда преимущества GraphQL?» Давайте рассмотрим их.

Избыточная и недостаточная выборка

Под *избыточной выборкой* подразумевается шаблон, обычно характерный для REST, когда в ответ на запрос по сети передаются ненужные и неиспользуемые данные. REST моделирует ресурсы, поэтому, выполняя запрос GET, скажем, ресурса `/movie/tt0105265`, REST API возвращает представление этого конкретного фильма – ни больше, ни меньше (листинг 1.4).

Листинг 1.4. Ответ REST API на запрос GET ресурса `/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
}
```

Но что, если приложение должно отображать только название и год фильма? В таком случае получается, что мы без необходимости передали слишком много данных. Кроме того, получение значений некоторых из этих полей может потребовать существенных затрат вычислительных ресурсов. Например, представьте, что для вычисления значения `averageReviews` в каждом запросе требуется проанализировать все отзывы о фильмах, хотя мы не показываем его в своем приложении. Эти избыточные вычисления потребуют много времени, что негативно скажется на производительности API. (Конечно, в реальном мире можно кешировать результаты таких расчетов, но это добавляет дополнительную сложность.) Точно так же *недостаточная выборка* – это шаблон, характерный для REST, когда запрос возвращает недостаточно данных.

Допустим, наше приложение должно отображать имя каждого актера, снявшегося в фильме. Сначала мы делаем запрос GET, чтобы получить ресурс `/movie/tt0105265`. Как было показано выше, в ответе мы получим массив идентификаторов актеров, снявшихся в этом фильме. Затем, чтобы получить необходимые данные, приложение должно в цикле обойти этот массив идентификаторов и получить имя каждого из них, выполнив дополнительные запросы к API:

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

При использовании GraphQL клиент может четко указать, какие данные он запрашивает, соответственно, всю необходимую информацию можно получить одним запросом, решив тем самым проблемы избыточной и недостаточной выборки. Это способствует повышению производительности на стороне сервера, потому что тратится меньше вычислительных ресурсов в слое хранения данных, по сети передается меньше данных и уменьшается общая задержка благодаря возможности получить все данные одним запросом.

Спецификация GraphQL

GraphQL – это спецификация, определяющая порядок взаимодействий клиент–сервер и описывающая возможности языка запросов GraphQL API. Наличие спецификации дает четкое руководство по реализации GraphQL API и четко определяет, какой API является GraphQL API, а какой нет.

REST не имеет спецификации, и существует множество различных реализаций, от REST-подобных до гипермедийных, таких как механизм управления состоянием приложения (HATEOAS). Наличие спецификации GraphQL упрощает обсуждение конечных точек, кодов состояния и документации. Все это встроено в GraphQL, что приводит к повышению производительности труда разработчиков и дизайнеров API. Спецификация ясно определяет путь, каким должны двигаться разработчики API.

В GraphQL все сущее – это графы

REST моделирует данные в виде ресурсов, но большинство взаимодействий с API осуществляются с точки зрения отношений. Например, в предыдущем запросе мы просим вернуть всех актеров, снявшихся в нем, и для каждого актера – все другие фильмы, в которых они снялись. Фактически мы запрашиваем отношения между актерами и фильмами. Эта идея отношений еще более заметна в реальных приложениях, где приходится обрабатывать отношения между клиентами и товарами в их заказах или пользователей и их сообщения в контексте диалогов.

GraphQL также может помочь объединить данные из разрозненных систем. Поскольку GraphQL не зависит от типов используемых хранилищ данных, есть возможность создавать GraphQL API, объединяющие данные из нескольких служб и реализующие недвусмысленный способ интеграции данных из разных систем в единую унифицированную схему GraphQL.

GraphQL также можно использовать для структурирования данных в шаблоне взаимодействия на основе компонентов. Поскольку каждый запрос GraphQL точ-

но описывает порядок обхода графа и возвращаемые поля, инкапсуляция этих запросов в компонентах приложения помогает упростить разработку и тестирование кода. Как это применяется на практике, будет показано в главе 5, где мы приступим к созданию приложения на основе библиотеки React.

Интроспекция

Интроспекция (или самоанализ) – это мощная особенность GraphQL, позволяющая запрашивать у GraphQL API поддерживаемые типы данных и запросы. Интроспекция – это один из способов самодокументирования API. Инструменты, использующие интроспекцию, позволяют получить документацию API в удобочитаемом виде, а инструменты визуализации – генерировать код для создания клиентов API.

1.2.4. *Недостатки GraphQL*

Конечно, GraphQL – это не панацея, и не следует думать об этой технологии как о решении всех проблем, связанных с API. Один из наиболее заметных недостатков GraphQL – некоторые общеизвестные методы, используемые в REST, неприменимы к GraphQL. Например, чтобы сообщить об успехе или неудаче, в REST обычно используются коды состояния HTTP. Код *200 OK* означает успешную обработку запроса, а *401 Not Authorized* означает, что мы забыли отправить токен авторизации или у нас нет разрешения на доступ к запрашиваемому ресурсу. В GraphQL каждый запрос возвращает *200 OK*, независимо от успеха обработки запроса. По этой причине ошибки в мире GraphQL приходится обрабатывать по-другому. Вместо одного кода состояния, описывающего результат запроса, в GraphQL ошибки обычно возвращаются в полях данных. Это означает, что в ответе на запрос GraphQL часть полей будут заполнены успешно, а часть – содержать признаки ошибки и должны обрабатываться соответственно.

Кеширование – еще один хорошо изученный аспект REST, который в GraphQL обрабатывается немного иначе. В REST есть возможность кешировать результат запроса к ресурсу `/movie/123` и всегда возвращать один и тот же точный результат в ответ на каждый запрос GET. Такое невозможно в GraphQL, потому что каждый запрос может содержать уникальный набор элементов, а это означает, что нельзя просто вернуть кешированный результат для всего запроса. Этот недостаток смягчается возможностью реализации кеширования на стороне клиента, даже притом что на практике большую часть времени запросы GraphQL выполняются в аутентифицированной среде, где кеширование неприменимо.

Другая проблема заключается в произвольной сложности запросов, конструируемых клиентом. Если клиент может составлять запросы по своему усмотрению, то как можно гарантировать, что они не станут слишком сложными и не окажут существенного влияния на производительность серверной инфраструктуры?

К счастью, GraphQL позволяет ограничить глубину запросов и дополнительно определить белый список запросов, которые могут выполняться (известный как список постоянных запросы). Однако с этим связана проблема реализации ограничения частоты запросов. В REST легко можно ограничить количество запросов, обрабатываемых за определенный период времени. В GraphQL эта задача усложняется, потому что клиент может запрашивать несколько объектов в одном за-

просе. Это приводит к необходимости реализации механизмов оценки затрат на обработку запросов.

Наконец, в GraphQL существует так называемая проблема $n + 1$ запросов, которая может вызвать множество обращений к хранилищу и негативно повлиять на производительность. Представьте, что мы запрашиваем информацию о фильме и всех актерах, снявшихся в нем. В базе данных для каждого фильма хранится список идентификаторов актеров и возвращается вместе со сведениями о фильме. Чтобы получить имена актеров, нам придется для каждого выполнить отдельный запрос к базе данных, что в сумме даст n (количество актеров) + 1 (сам фильм) запросов к базе данных. Для решения проблемы $n + 1$ запросов можно использовать такие инструменты, как DataLoader, позволяющие группировать и кешировать запросы к базе данных, повышая производительность. Другой подход к решению проблемы $n + 1$ запросов – использовать механизмы интеграции GraphQL с базами данных, таких как библиотека Neo4j GraphQL и PostGraphile, позволяющие генерировать один запрос к базе данных на основе произвольного запроса GraphQL и гарантирующие выполнение единственного обращения к базе данных.

> **Ограничения GraphQL**
>
> Говоря о базах данных, важно понимать, что GraphQL – это язык запросов к API, а не к базе данных. В GraphQL отсутствует поддержка сложных операций, имеющаяся в языках запросов к базам данных, таких как агрегирование, проецирование и обход путей переменной длины.

1.2.5. *Инструменты GraphQL*

В этом разделе мы рассмотрим некоторые инструменты, характерные для GraphQL, которые помогут нам создавать, тестировать и запрашивать GraphQL API. Эти инструменты используют механизм интроспекции в GraphQL, позволяя извлекать схему развернутой конечной точки GraphQL, создавать документацию, проверять запросы, поддерживать функцию автоматического дополнения и т. д.

GraphiQL

GraphiQL – это инструмент, встраиваемый в браузер и предназначенный для изучения GraphQL API и отправки запросов ему. С помощью GraphiQL можно посылать запросы к GraphQL API и просматривать результаты. Благодаря механизму интроспекции в GraphQL можно просматривать типы, поля и запросы, поддерживаемые GraphQL API. Кроме того, благодаря особенностям системы типов в GraphQL есть возможность немедленной проверки запросов при их создании. GraphiQL – это пакет с открытым исходным кодом, который теперь поддерживается GraphQL Foundation. GraphiQL распространяется и в форме автономного инструмента, и в форме компонента React, поэтому он часто встраивается в более крупные веб-приложения (рис. 1.5).

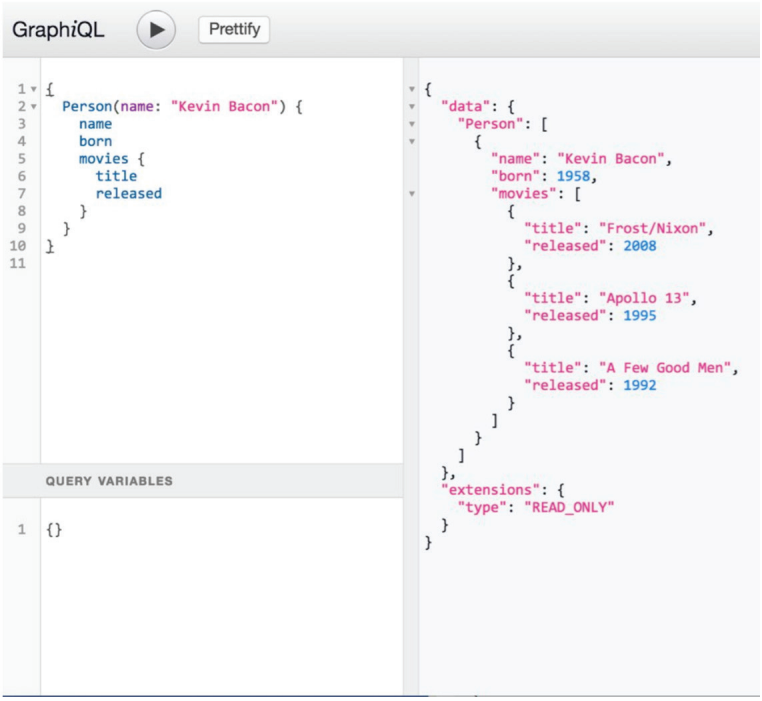

Рис. 1.5. Окно **GraphiQL**

GraphQL Playground

GraphQL Playground, как и GraphiQL, – это инструмент, встраиваемый в браузер и предназначенный для выполнения запросов, просмотра результатов и изучения схемы GraphQL API (рис. 1.6). GraphQL Playground имеет несколько дополнительных функций, таких как просмотр определений типов, поиск по схеме GraphQL и простое добавление заголовков запросов (например, необходимых для аутентификации). Когда-то GraphQL Playground был включен по умолчанию в некоторые серверные реализации, такие как Apollo Server; однако с тех пор он устарел и его развитие остановилось. Я упомянул GraphQL Playground только потому, что он все еще развернут во многих конечных точках GraphQL и вы можете столкнуться с ним в какой-то момент.

Apollo Studio

Apollo Studio – облачная платформа компании Apollo, поддерживающая возможности создания, проверки и защиты GraphQL API (рис. 1.7). Я включил Apollo Studio в этот раздел, потому что функция *Explorer* в Apollo Studio аналогична инструментам GraphiQL и GraphQL Playground, упомянутым выше, и позволяет создавать и выполнять операции GraphQL. Кроме того, Explorer в Apollo Studio по умолчанию используется в Apollo Server (начиная с версии 3), поэтому в этой книге мы будем использовать Apollo Studio для выполнения операций с нашим GraphQL API в процессе его разработки.

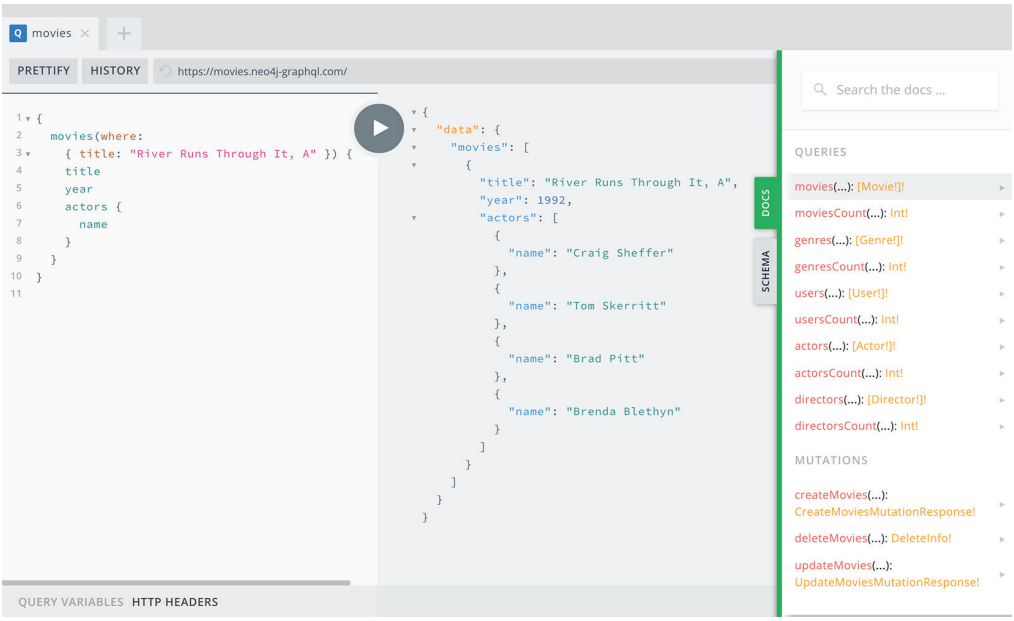

Рис. 1.6. Окно **GraphQL Playground**

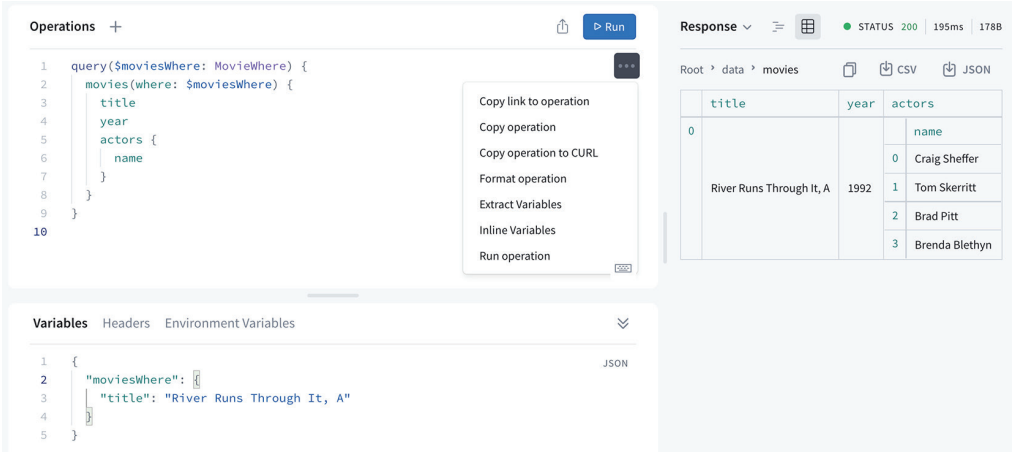

Рис. 1.7. Окно **Apollo Studio**

1.3. React

React – декларативная библиотека, предназначенная для создания пользовательских интерфейсов на JavaScript. React использует виртуальную модель DOM (копию фактической объектной модели документа) для эффективного выявления обновлений в модели DOM, необходимых для отображения представлений по мере изменения состояния приложения и данных. Используя React, пользователи просто создают представления, отображающие данные приложения, а библиотека эффективно обрабатывает обновления DOM. Компоненты библиотеки реализуют

логику обработки данных и отображения пользовательского интерфейса, не раскрывая своей внутренней структуры, поэтому их легко комбинировать для создания сложных пользовательских интерфейсов и приложений.

1.3.1. *Компоненты React*

Рассмотрим простой компонент React в листинге 1.5.

Листинг 1.5. Простой компонент React

```
import React, { useState } from "react";

function MovieTitleComponent(props) {
  const [movieTitle, setMovieTitle] = useState(
    "River Runs Through It, A"
  );

  return <div>{movieTitle}</div>
}
export default MovieTitleComponent;
```

Импорт библиотеки React и обработчика `useState` **для управления переменными состояния**

Наш компонент – это функция, получающая значения `props` **из других компонентов, стоящих выше в иерархии компонентов React**

С использованием обработчика `useState` **создаются новая переменная состояния и связанная с ней функция обновления этого значения**

Здесь мы получаем доступ к значению `movieTitle` **из состояния компонента и визуализируем его внутри тега** `div`

Экспорт компонента, чтобы его можно было использовать в других компонентах React

Библиотеки компонентов

Компоненты реализуют логику обработки данных и отображения пользовательского интерфейса и легко компонуются друг с другом, поэтому есть возможность создавать и распространять библиотеки компонентов и подключать их как зависимости к проектам, что упрощает использование сложных стилей пользовательского интерфейса. Обсуждение применения библиотек компонентов выходит за рамки этой книги, однако хорошим примером вам может послужить библиотека Material UI, включающая многие популярные компоненты пользовательского интерфейса, такие как сетчатые макеты, таблицы данных, элементы навигации и ввода.

1.3.2. *JSX*

React обычно используется с расширением языка JavaScript под названием JSX. JSX похож на XML и позволяет описывать пользовательские интерфейсы и объединять компоненты React. React можно использовать и без JSX, но многим разработчикам нравятся простота и ясность кода на JSX. Мы познакомимся с JSX в главе 5, где еще рассматриваются некоторые другие концепции React, такие как однонаправленный поток данных, свойства и состояние, а также выборка данных с помощью React.

1.3.3. *Инструменты React*

Ниже перечисляются некоторые полезные инструменты, помогающие проектировать, создавать и отлаживать приложения React. Для разработки приложений

React существует целая экосистема инструментов, поэтому не стоит считать этот список исчерпывающим.

Create React App

Create React App – это инструмент командной строки, позволяющий быстро создать каркас приложения React, включая настройки параметров сборки, установку зависимостей и создание начального шаблона приложения. Мы будем использовать Create React App в главе 5, когда займемся разработкой пользовательского интерфейса нашего приложения.

React Chrome Devtools

React DevTools – это расширение браузера, позволяющее исследовать приложение React, иерархию его компонентов, а также свойства и состояние каждого компонента во время работы приложения, что здорово помогает в отладке. Иногда очень полезно иметь возможность видеть, как организованы компоненты в различных сценариях использования (рис. 1.8).

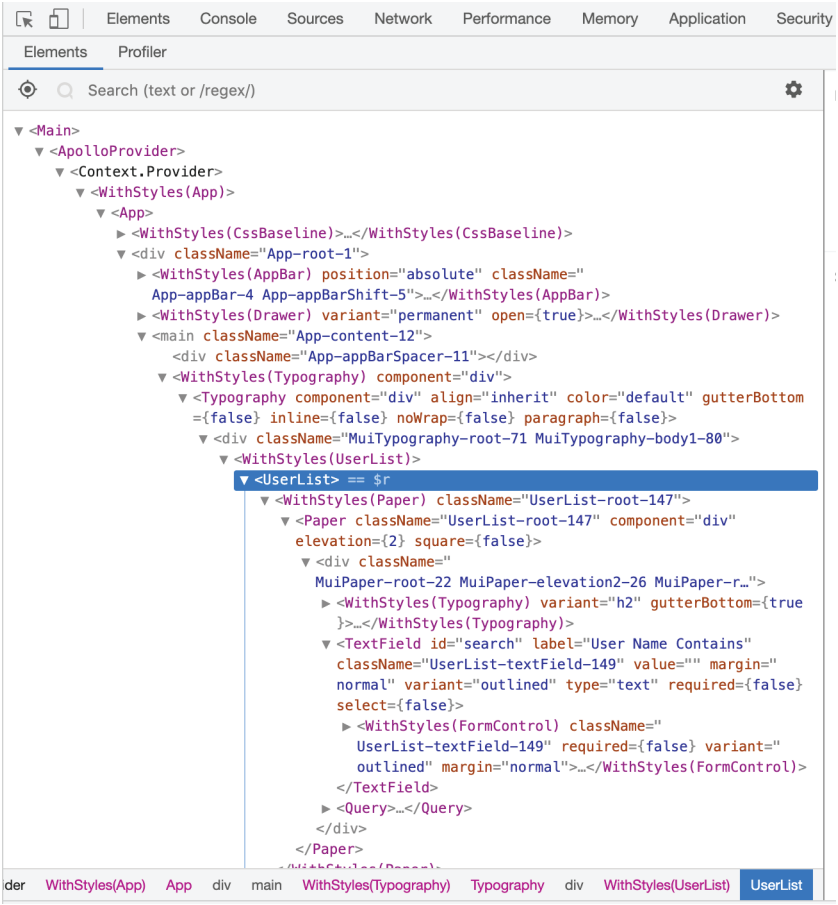

Рис. 1.8. React Chrome DevTools

1.4. Apollo

Apollo – это набор инструментов, упрощающих использование GraphQL, в том числе на сервере, клиенте и в облаке. Для создания нашего GraphQL API мы будем использовать Apollo Server и библиотеку Node.js, а для создания и отправки запросов к GraphQL API из приложения – Apollo Client, клиентскую библиотеку на JavaScript и Apollo Studio Explorer.

1.4.1. *Apollo Server*

Apollo Server позволяет развернуть сервер Node.js, обслуживающий конечную точку GraphQL, путем определения типов и функций разрешения. Apollo Server можно использовать со многими веб-фреймворками, однако по умолчанию чаще всего используется Express.js. Apollo Server также можно использовать с бессерверными функциями, такими как Amazon Lambda и Google Cloud Functions. Установить Apollo Server можно с помощью `npm`: `npm install apollo-server`.

1.4.2. *Apolo Client*

Apollo Client – это библиотека на JavaScript для выполнения запросов к GraphQL API. Она интегрируется со многими фреймворками для разработки пользовательских интерфейсов, включая React и Vue.js, а также с собственными мобильными версиями для iOS и Android. Мы будем использовать React Apollo Client в компонентах React для выборки данных из GraphQL. Apollo Client поддерживает кеширование клиентских данных и может использоваться для управления локальными данными состояния. Библиотеку React Apollo можно установить с помощью `npm`: `npm install @apollo/client`.

1.5. База данных Neo4j

Neo4j – это графовая база данных с открытым исходным кодом. В отличие от других баз данных, основанных на таблицах или документах, в Neo4j используется модель данных в виде графа, известная также как *графовая модель свойств*, позволяющая моделировать, хранить и запрашивать данные в виде графа. Графовые базы данных, такие как Neo4j, оптимизированы для работы с графами и обхода сложных графов, например.

Одно из преимуществ использования графовой базы данных – поддержка одной и той же модели данных в виде графа во всем приложении: и в пользовательском интерфейсе, и на сервере, и в базе данных. Еще одно преимущество – более высокая эффективность графовых баз данных по сравнению, например, с реляционными. Многие запросы GraphQL включают внутренние подзапросы – это эквивалент операции `JOIN` в реляционной базе данных. Базы данных графов оптимизированы для эффективного выполнения таких операций и поэтому идеально подходят для использования в серверной части GraphQL API.

> **ПРИМЕЧАНИЕ.** Важно отметить, что при работе с GraphQL мы не обращаемся к базе данных напрямую. GraphQL поддерживает интеграцию с базами данных, но, вообще говоря, GraphQL API – это слой, находящийся между нашим приложением и базой данных.

1.5.1. *Графовая модель свойств*

Подобно многим графовым базам данных, Neo4j использует графовую модель свойств (рис. 1.9). Вот некоторые компоненты этой модели:

- узлы – сущности или объекты в модели данных;
- отношения – соединения между узлами;
- метки – семантика группировки узлов;
- свойства – пары ключ/значение, хранящиеся в узлах и отношениях.

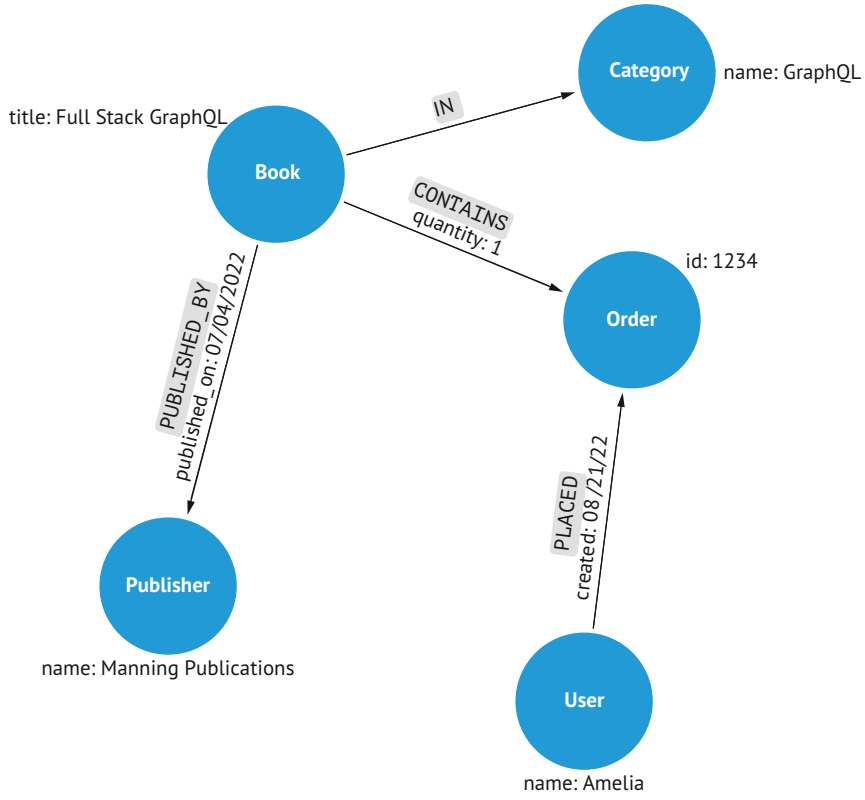

Рис. 1.9. Пример графа свойств книг, издателей, клиентов и заказов

Графовая модель свойств позволяет гибко выражать сложные и взаимосвязанные данные. Эта модель данных имеет еще одно преимущество – она точнее отражает наше мысленное представление данных при работе с предметной областью.

1.5.2. *Язык запросов Cypher*

Cypher – это декларативный язык запросов к графовым базам данных, таким как Neo4j, и механизмам графовых вычислений. Cypher можно сравнить с языком SQL, только он предназначен для работы не с таблицами, а с графовыми данными. Главная особенность Cypher – возможность *сопоставления с шаблоном*. В листинге 1.6 показан простой запрос на языке Cypher, возвращающий фильмы и актеров, снявшихся в этих фильмах.

Листинг 1.6. Простой запрос на языке Cypher, возвращающий фильмы и актеров, снявшихся в них

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

Описание шаблона графа для поиска данных

Возврат данных, соответствующих описанному шаблону графа

В этом запросе Cypher за инструкцией `MATCH` следует описание шаблона графа. В этом шаблоне узлы определяются в круглых скобках, например `(m:Movie)`. Здесь `:Movie` указывает, что узлы должны сопоставляться с меткой `Movie`, а `m` перед двоеточием – это имя переменной, которая связывается с любыми узлами, соответствующими шаблону. На переменную `m` можно ссылаться далее в запросе.

Отношения определяются квадратными скобками (например, `<-[r:ACTED_IN]-`) и подчиняются аналогичному соглашению, где `:ACTED_IN` объявляет тип отношения `ACTED_IN`, а `r` – это переменная, представляющая любое отношение, соответствующее шаблону, на которую можно ссылаться далее в запросе.

В инструкции `RETURN` указываются данные, возвращаемые запросом. Здесь указаны переменные `m`, `r` и `a`, объявленные в инструкции `MATCH` и привязанные к узлам и отношениям в базе данных, соответствующим элементам шаблона графа.

1.5.3. *Инструменты Neo4j*

Для управления экземплярами Neo4j мы будем использовать Neo4j Desktop – инструмент разработчика для отладки запросов и взаимодействия с базой данных Neo4j. Для отправки запросов в Neo4j из GraphQL API мы будем использовать клиентский драйвер JavaScript Neo4j, а также библиотеку Neo4j GraphQL, обеспечивающую интеграцию Node.js GraphQL с Neo4j.

Neo4j Desktop

Neo4j Desktop – это центр управления Neo4j (рис. 1.10). Из Neo4j Desktop можно управлять экземплярами базы данных Neo4j, изменять их настройки, устанавливать плагины и приложения (например, инструменты визуализации), а также выполнять административные функции, такие как ввод/вывод содержимого базы данных. Neo4j Desktop часто используется для управления Neo4j и доступен для загрузки по адресу **https://neo4j.com/download/**.

Neo4j AuraDB

Neo4j AuraDB – это полностью управляемая облачная служба Neo4j, предлагающая экземпляры Neo4j в облаке. Есть тариф AuraDB с бесплатным обслуживанием, что делает эту службу отличным вариантом для разработки и развертывания любительских проектов. Более подробно мы поговорим о Neo4j AuraDB в главе 8, когда будем изучать развертывание нашего приложения полного цикла в облаке. Подписаться на бесплатное обслуживание в Neo4j AuraDB можно на странице **https://neo4j.com/cloud/platform/aura-graph-database/**.

Neo4j Browser

Neo4j Browser – встроенная в браузер инструментальная среда запросов и один из основных способов взаимодействия с Neo4j во время разработки (рис. 1.11).

С помощью Neo4j Browser можно посылать базе данных запросы Cypher и отображать результаты в виде графов или таблиц.

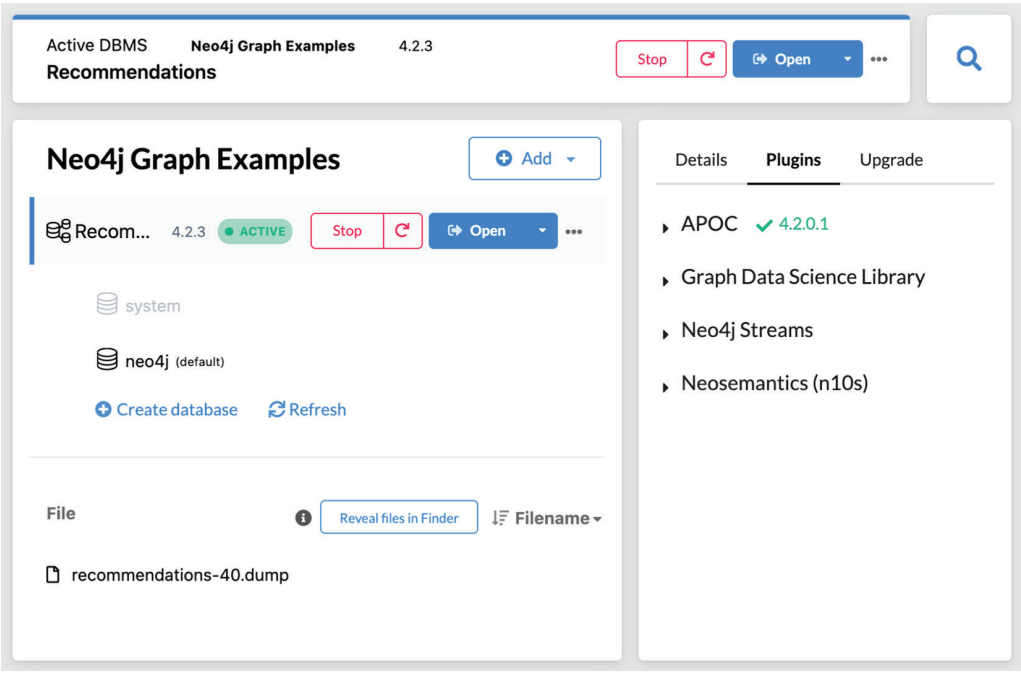

Рис. 1.10. Neo4j Desktop

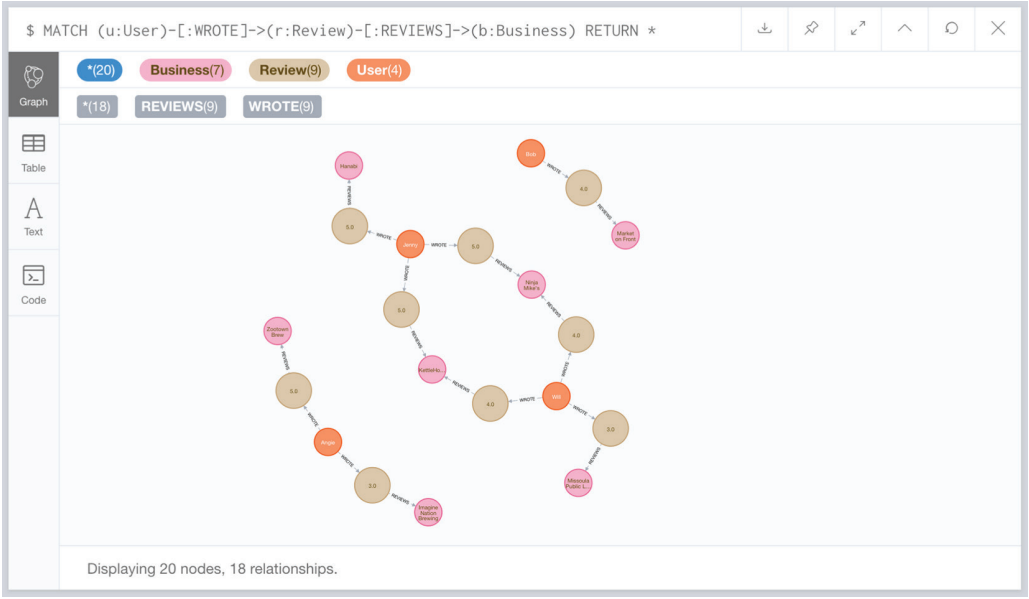

Рис. 1.11. Neo4j Browser

Клиентские драйверы Neo4j

Наша конечная цель – создание приложения, взаимодействующего с базой данных Neo4j, поэтому мы будем использовать клиентские драйверы Neo4j. Драйверы доступны на многих языках программирования (Java, Python, .Net, JavaScript, Go и т. д.), но мы будем использовать драйвер Neo4j JavaScript.

> **ПРИМЕЧАНИЕ.** Драйвер Neo4j JavaScript имеет две версии: для Node.js и для браузера (что позволяет подключаться к базе данных непосредственно из браузера), однако в этой книге мы будем использовать только версию для Node.js.

Драйвер Neo4j JavaScript можно установить с помощью `npm`:

```
npm install neo4j-driver
```

В листинге 1.7 показан пример использования драйвера Neo4j JavaScript для выполнения запроса Cypher и вывода результатов

Листинг 1.7. Простой пример использования драйвера Neo4j JavaScript

```
const neo4j = require("neo4j-driver");          ← Импорт модуля neo4j-driver

const driver = neo4j.driver("neo4j://localhost:7687",   ← Создание экземпляра драйвера с указанием строки подключения к базе данных
  neo4j.auth.basic("neo4j", "letmein"));        ← Имя пользователя и пароль для доступа к базе данных

const session = driver.session();               ← Для выполнения определенной работы должны создаваться сеансы

session.run("MATCH (n) RETURN COUNT(n) AS num")   ← Запуск запроса в транзакции с автоматическим подтверждением; он возвращает объект Promise
  .then(result => {                             ← Promise разрешается в набор результатов
    const record = result.records[0];           ← Выбор первой записи из набора результатов
    console.log(`Your database has ${record['num']} nodes`);
  })
  .catch(error => {
    console.log(error);
  })
  .finally( () => {                             ← Не забывайте закрывать сеансы!
    session.close();
  )
```

Мы будем использовать драйвер Neo4j JavaScript в наших функциях разрешения как один из способов выборки данных в GraphQL API.

Библиотека Neo4j GraphQL

Библиотека Neo4j GraphQL – это слой, преобразующий запросы GraphQL в Cypher и пересылающий их в Neo4j. Она может работать с любой реализацией сервера JavaScript GraphQL, например с Apollo. Далее в книге мы узнаем, как использовать эту библиотеку для:

1. определения типов GraphQL и управления схемой базы данных Neo4j;
2. создания полного набора операций CRUD в GraphQL API, исходя из определений типов;
3. генерирования единственного запроса к базе данных на языке Cypher на основе произвольных запросов GraphQL (решение проблемы $n + 1$ запросов);
4. добавления пользовательской логики в GraphQL API с помощью Cypher.

GraphQL не зависит от слоя хранения данных – GraphQL API можно реализовать с применением любого источника данных или базы данных, – но использование графовой базы данных дает дополнительные преимущества, такие как сокращение операций отображения и преобразования данных и оптимизация производительности для обхода сложных графов. Библиотека Neo4j GraphQL помогает создавать GraphQL API, основанные на графовой базе данных Neo4j. Знакомство с библиотекой Neo4j GraphQL мы начнем в главе 4, а дополнительную информацию о ней можно найти на странице **https://neo4j.com/product/graphql-library/**.

1.6. Как все это сочетается

Теперь, рассмотрев отдельные части стека GraphQL, посмотрим, как они сочетаются в контексте приложения полного цикла, а в качестве примера используем приложение поиска фильмов. Наше воображаемое приложение имеет три простых требования:

1. позволяет искать фильмы по названию;
2. отображает найденные фильмы вместе с дополнительными сведениями о них, такими как рейтинг или жанр;
3. выводит список похожих фильмов, которые могут быть хорошей рекомендацией, если пользователю понравился найденный фильм.

На рис. 1.12 показано, как различные компоненты будут сочетаться друг с другом в потоке, включающем отправку запроса клиентским приложением в GraphQL API, анализ информации, полученной из базы данных Neo4j, ее передачу обратно к клиенту и отображение результатов в пользовательском интерфейсе.

1.6.1. *React и Apollo Client: выполнение запроса*

Пользовательский интерфейс приложения реализован на React; в частности, в нем есть компонент `MovieSearch`, отображающий текстовое поле для ввода названия искомого фильма. Этот компонент также содержит логику объединения пользовательского ввода с запросом GraphQL и отправки этого запроса на сервер GraphQL для обработки с использованием интеграции Apollo Client React. В листинге 1.8 показано, как мог бы выглядеть запрос GraphQL для поиска фильма с названием «River Runs Through It» («Там, где течет река»).

Листинг 1.8. Запрос GraphQL для поиска фильма с названием «River Runs Through It»

```
{
  moviesByTitle(title: "River Runs Through It") {
    title
```

```
    poster
    imdbRating
    genres {
      name
    }
    recommendedMovies {
      title
      poster
    }
  }
}
```

Логика выборки данных реализуется клиентом Apollo Client, который используется компонентом `MovieSearch`. Apollo Client обеспечивает кеширование данных, поэтому, когда пользователь вводит поисковый запрос, Apollo Client сначала проверяет кеш. Если прежде этот запрос GraphQL не обрабатывался, то он отправляется на сервер GraphQL в виде HTTP-запроса POST к конечной точке `/graphql`.

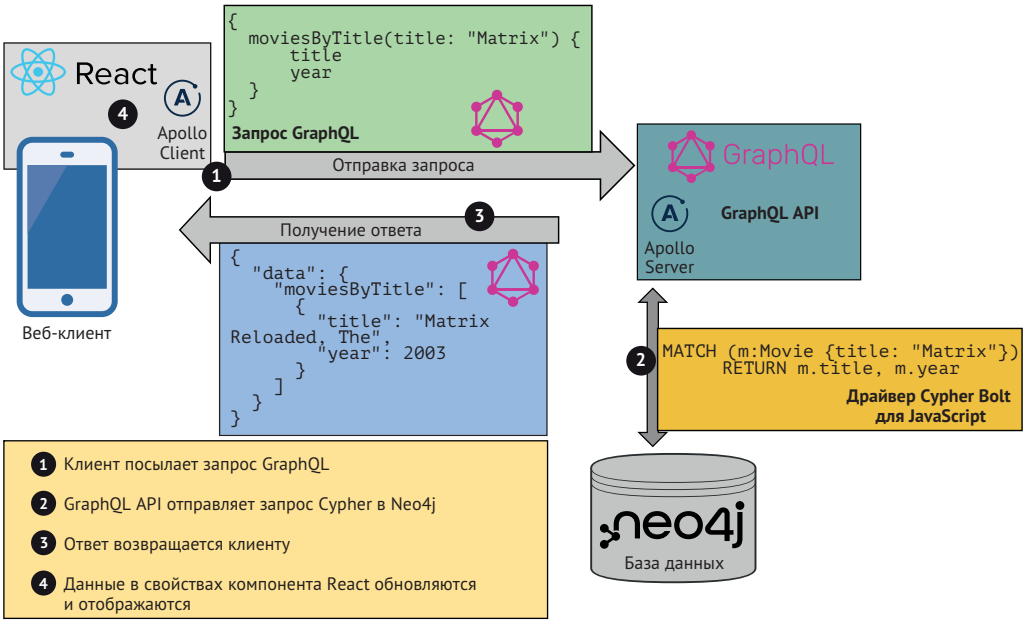

Рис. 1.12. Выполнение запроса на поиск фильма через приложение GraphQL полного цикла

1.6.2. *Apollo Server и серверная часть GraphQL*

Серверная часть нашего приложения – это приложение Node.js, использующее Apollo Server и библиотеку Express для обслуживания конечной точки `/graphql` через HTTP. Сервер GraphQL состоит из сетевого слоя, отвечающего за обработку HTTP-запросов, извлечение операции GraphQL и возврат HTTP-ответов, и схемы GraphQL, которая определяет точки входа и структуры данных для API и реализует разрешение данных, полученных из хранилища, выполняя функции разрешения.

Когда Apollo Client посылает запрос, сервер GraphQL обрабатывает его, проверяя запрос, а затем приступает к разрешению, сначала вызывая корневую функцию разрешения, в данном случае `Query.moviesByTitle`. Эта функция разрешения получает аргумент `title` – значение, введенное пользователем в текстовое поле поиска. Внутри функции разрешения находится логика, выполняющая запрос к базе данных, выбирающая фильмы с названиями, соответствующими поисковому запросу, дополнительные сведения о них и отыскивающая список других фильмов, похожих на найденный.

Реализация функции разрешения

В данной книге мы покажем два способа реализации функций разрешения:

- прямолинейный способ определения запросов к базе данных;
- автоматическое создание функций разрешения с использованием библиотек GraphQL, таких как библиотека Neo4j GraphQL.

В этом примере используется только первый способ.

Функции разрешения выполняются каскадно (рис. 1.13) – в данном случае первой вызывается функция разрешения поля запроса `movieByTitle` – корневая функция разрешения для данной операции. `movieByTitle` возвращает список фильмов, а затем вызываются функции разрешения для каждого поля, присутствующего в запросе с передачей им элементов из списка фильмов, полученного функцией `movieByTitle`, – по сути выполняется обход этого списка фильмов.

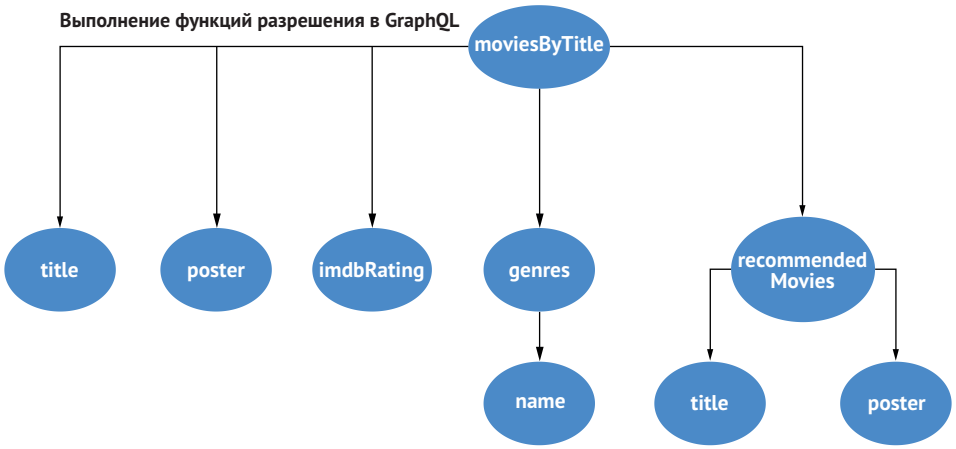

Рис. 1.13. Функции разрешения GraphQL вызываются каскадно

Каждая функция реализует логику разрешения данных для своей части общей схемы GraphQL. Например, функция `recommendedMovies` отыскивает похожие фильмы, которые также могут понравиться зрителю. В нашем случае для этого выполняется простой запрос Cypher к базе данных, отыскивающий пользователей, просмотревших данный фильм, и другие фильмы, просмотренные из которых затем формируются в рекомендации методом коллаборативной (коллективной) фильт-

рации, как показано в листинге 1.9. Этот запрос передается в базу данных Neo4j с помощью клиентского JavaScript-драйвера Neo4j для Node.js.

Листинг 1.9. Простой запрос Cipher для выбора рекомендуемых фильмов

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

Проблема *n* + 1 запросов

В этом примере наглядно проявляется проблема *n* + 1 запросов. Корневая функция разрешения возвращает список фильмов. Теперь, чтобы разрешить запрос GraphQL, нужно вызвать функцию разрешения `actor` один раз для каждого фильма. В результате базе данных будет отправлено множество запросов, что может отрицательно сказаться на производительности.

В идеале желательно выполнить только один запрос к базе данных, извлекающий все данные, необходимые для разрешения запроса GraphQL. Эта задача имеет несколько решений:

- группировать запросы с помощью библиотеки DataLoader;
- использовать библиотеки GraphQL, такие как Neo4j GraphQL, чтобы сгенерировать один запрос к базе данных на основе произвольного запроса GraphQL, используя графовую природу GraphQL, и тем самым избежать негативного влияния на производительность выполнением нескольких запросов к базе данных.

1.6.3. *React и Apollo Client: обработка ответа*

После завершения выборки и отправки данных клиенту Apollo Client клиент обновляет свой кеш, и поэтому при попытке повторно выполнить этот же поисковый запрос клиент вернет данные из кеша, без обращения к серверу GraphQL.

Наш React-компонент `MovieSearch` передает полученные результаты компоненту `MovieList` в виде свойства, а тот, в свою очередь, отображает серию компонентов `Movie` со сведениями о фильмах – в данном случае результат содержит только один фильм. И на экране наш пользователь видит список результатов поиска (рис. 1.14)!

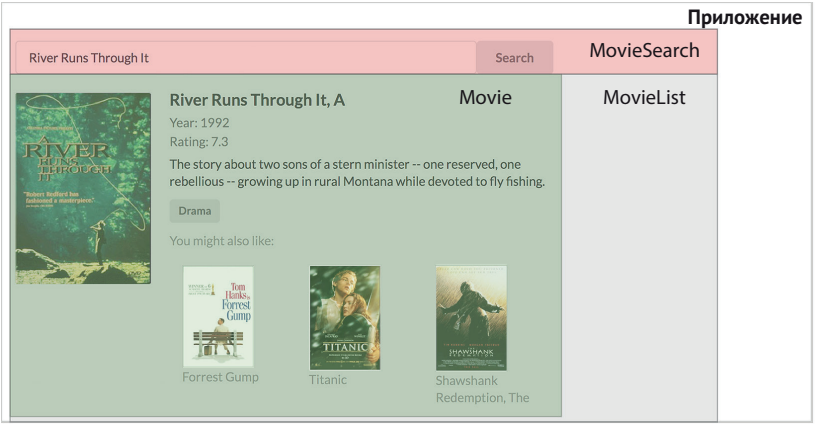

Рис. 1.14. Все вместе компоненты React создают сложный пользовательский интерфейс

Цель этого примера – показать, как используются GraphQL, React, Apollo и база данных Neo4j для создания простого приложения полного цикла. Мы опустили многие детали, такие как аутентификация, авторизация и оптимизация производительности, но не волнуйтесь, мы подробно рассмотрим все это далее в книге!

1.7. Что мы будем строить в этой книге

Мы надеемся, что простой пример приложения поиска фильмов в этой главе стал достойным введением в концепции, которые будут изучаться на протяжении всей книги. Но давайте начнем с нуля и создадим новое приложение – приложение обзора компаний, – проработав его требования и дизайн GraphQL API, и постепенно накопим знания о GraphQL. Для демонстрации концепций, описанных в этой книге, мы создадим веб-приложение, использующее GraphQL, React, Apollo и Neo4j. Вот основные требования к нему:

- выводить перечень компаний и дополнительные сведения о них;
- давать пользователям возможности писать отзывы о компаниях;
- давать пользователям возможности искать компании и показывать персональные рекомендации.

Чтобы реализовать это приложение, нам потребуется спроектировать и реализовать соответствующий GraphQL API, пользовательский интерфейс и базу данных. Также мы должны будем решить такие проблемы, как аутентификация и авторизация, и в конце развернуть приложение в облаке.

1.8. Упражнения

1. Для более полного знакомства с GraphQL и запросами GraphQL изучите исходный код приложения поиска фильмов, доступный по адресу **https://movies.neo4j-graphql.com/**. Откройте URL в веб-браузере, чтобы получить доступ к GraphQL Playground, и исследуйте содержимое вкладок `DOCS` и `SCHEMA`, где приводятся определения типов.

   Попробуйте написать запросы, используя следующие подсказки:

   - найдите наименования первых 10 фильмов, отсортированные по названиям;
   - кто снимался в фильме «Jurassic Park» («Парк Юрского периода»)?
   - к каким жанрам относится фильм «Jurassic Park»? Какие еще фильмы есть в этих жанрах?
   - найдите фильм с самым высоким рейтингом `imdbRating`.

2. Поразмышляйте над приложением обзора компаний, представленным выше в этой главе. Сможете ли вы создать определения типов GraphQL, необходимые для этого приложения?
3. Загрузите Neo4j и ознакомьтесь с инструментами Neo4j Desktop и Neo4j Browser. Изучите пример набора данных Neo4j Sandbox, доступный по адресу **https://neo4j.com/sandbox/**.

Решения упражнений, а также примеры кода из этой книги можно найти в репозитории GitHub: **https://github.com/johnymontana/fullstack-graphql-book**.

Итоги

- GraphQL – это язык запросов к API и среда выполнения запросов. GraphQL можно использовать с любыми хранилищами данных. Чтобы создать GraphQL API, сначала нужно определить типы, включая поля и взаимосвязи между ними, а также описать граф данных.
- React – это библиотека на JavaScript для создания пользовательских интерфейсов. Для создания компонентов, инкапсулирующих данные и логику, используется расширение JSX. Компоненты можно комбинировать, что позволяет создавать сложные пользовательские интерфейсы.
- Apollo – это коллекция инструментов для работы с GraphQL как на стороне клиента, так и на стороне сервера. Apollo Server – это библиотека Node.js для создания GraphQL API. Apollo Client – это клиент JavaScript для GraphQL, интегрирующийся со многими веб-фреймворками, включая React.
- Neo4j – это графовая база данных с открытым исходным кодом, использующая графовую модель свойств, состоящую из узлов, взаимосвязей, меток и свойств. Для взаимодействия с Neo4j используется язык запросов Cypher.
- Все перечисленные технологии можно использовать вместе для создания приложений GraphQL полного цикла.