Разработка веб-приложений GraphQL с React, Node.js и Neo4j

Уильям Лион

MANNING

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Уильям Лион

Разработка веб-приложений GraphQL с *React*, *Node.js* и *Neo4j*

Full Stack GraphQL Applications

with REACT, NODE.JS, and NEO4J

William Lyon

MANNING
SHELTER ISLAND

Разработка веб-приложений GraphQL

с React, Node.js и Neo4j

Уильям Лион

ДМК Пресс
издательство
Москва, 2023

УДК 004.04
ББК 32.372
Л60

Л60 Уильям Лион
Разработка веб-приложений GraphQL с React, Node.js и Neo4j / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2023. – 262 с.: ил.

ISBN 978-5-93700-185-6

Эта книга научит вас создавать графовые веб-приложения с использованием технологии GraphQL, преимущества которой усиливают такие графовые инструменты и хранилища данных, как React, Apollo и Neo4j. Вначале вы познакомитесь с GraphQL и собственно с понятием графов, затем сосредоточитесь на разработке клиентского приложения с использованием React и, наконец, создадите и развернете в облаке полноценное веб-приложение, поддерживающее поиск, аутентификацию и многое другое.

Издание предназначено для веб-разработчиков полного цикла, заинтересованных в технологии GraphQL и имеющих базовое представление о Node.js API и особенностях клиентских приложений на JavaScript, использующих этот API. Опыт работы с GraphQL или графовыми базами данных не обязателен.

Copyright © DMK Press 2022. Authorized translation of the English edition © 2022 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материа,л, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-61729-703-8 (англ.)
ISBN 978-5-93700-185-6 (рус.)

Copyright © 2022 by Manning Publications Co.
© Оформление, перевод на русский язык, издание, ДМК Пресс, 2023

Оглавление

Предисловие от издательства 10
Вступление 11
Благодарности 12
О книге 13
Кому адресована эта книга 13
Организация книги 13
О примерах программного кода 14
Требования к программному обеспечению 14
Живое обсуждение книги 15
Другие онлайн-ресурсы 15
Об авторе 16
Об иллюстрации на обложке 16

Часть I. Введение в стек GraphQL 17

Глава 1. Что такое стек GraphQL? 18
1.1. Обзор стека GraphQL 18
1.2. GraphQL 20
1.2.1. Определения типов в GraphQL 20
1.2.2. Запросы GraphQL 22
1.2.3. Преимущества GraphQL 25
1.2.4. Недостатки GraphQL 27
1.2.5. Инструменты GraphQL 28
1.3. React 30
1.3.1. Компоненты React 31
1.3.2. JSX 31
1.3.3. Инструменты React 31
1.4. Apollo 33
1.4.1. Apollo Server 33
1.4.2. Apolo Client 33
1.5. База данных Neo4j 33
1.5.1. Графовая модель свойств 34
1.5.2. Язык запросов Cypher 34
1.5.3. Инструменты Neo4j 35
1.6. Как все это сочетается 38
1.6.1. React и Apollo Client: выполнение запроса 38
1.6.2. Apollo Server и серверная часть GraphQL 39
1.6.3. React и Apollo Client: обработка ответа 41

1.7. Что мы будем строить в этой книге ... 42
1.8. Упражнения ... 42
Итоги ... 43

Глава 2. Графовое мышление с GraphQL ... 44

2.1. Данные приложения – это граф ... 44
2.2. Графы в GraphQL ... 46
2.2.1. Моделирование API с применением определений типов: разработка на основе GraphQL ... 46
2.2.2. Выборка данных с помощью функций разрешения ... 53
2.2.3. Наша первая функция разрешения ... 54
2.3. Объединение определений типов и функций разрешения в Apollo Server ... 57
2.3.1. Apollo Server ... 57
2.3.2. Apollo Studio ... 57
2.3.3. Реализация функций разрешения ... 59
2.3.4. Выполнение запросов с помощью Apollo Studio ... 62
2.4. Упражнения ... 63
Итоги ... 63

Глава 3. Графы в базе данных ... 64

3.1. Обзор Neo4j ... 64
3.2. Моделирование графовых данных в Neo4j ... 65
3.2.1. Графовая модель свойств ... 66
3.2.2. Ограничения базы данных и индексы ... 69
3.3. Вопросы моделирования данных ... 69
3.3.1. Узел и свойство ... 69
3.3.2. Узел и отношение ... 70
3.3.3. Индексы ... 70
3.3.4. Специфика типов отношений ... 70
3.3.5. Выбор направления отношений ... 70
3.4. Инструменты: Neo4j Desktop ... 70
3.5. Инструменты: Neo4j Browser ... 71
3.6. Cypher ... 72
3.6.1. Сопоставление с образцом ... 72
3.6.2. Свойства ... 72
3.6.3. CREATE ... 73
3.6.4. MERGE ... 76
3.6.5. Определение ограничений на Cypher ... 77
3.6.6. MATCH ... 78
3.6.7. Агрегаты ... 79
3.7. Использование клиентских драйверов Neo4j ... 79
3.8. Упражнения ... 80
Итоги ... 80

Глава 4. Библиотека Neo4j GraphQL ... 81
4.1. Распространенные проблемы GraphQL ... 82
4.1.1. Низкая производительность и проблема *n* + 1 запросов ... 82
4.1.2. Типовой код и продуктивность разработчиков ... 82
4.2. Введение в средства интеграции GraphQL с базой данных ... 83
4.3. Библиотека Neo4j GraphQL ... 83
4.3.1. Настройка проекта ... 84
4.3.2. Генерирование схемы GraphQL из определений типов ... 88
4.4. Основы запросов GraphQL ... 90
4.5. Упорядочение и разбиение на страницы ... 93
4.6. Вложенные запросы ... 94
4.7. Фильтрация ... 95
4.7.1. Аргумент where ... 95
4.7.2. Вложенные фильтры ... 96
4.7.3. Логические операторы: AND, OR ... 97
4.7.4. Фильтрация выборки ... 98
4.8. Работа с датой/временем ... 100
4.8.1. Использование типа Date в запросах ... 100
4.8.2. Фильтры по полям с типами Date и DateTime ... 101
4.9. Работа с пространственными данными ... 102
4.9.1. Выборка данных типа Point ... 102
4.9.2. Фильтрация по расстояниям ... 103
4.10. Добавление своей логики в GraphQL API ... 104
4.10.1. Директива @cypher ... 104
4.10.2. Реализация собственных функций разрешения ... 108
4.11. Определение схемы GraphQL в существующей базе данных ... 110
4.12. Упражнения ... 111
Итоги ... 112

Часть II. Создание пользовательского интерфейса ... 113

Глава 5. Создание пользовательского интерфейса с помощью React ... 114
5.1. Обзор React ... 115
5.1.1. JSX и элементы React ... 115
5.1.2. Компоненты React ... 116
5.1.3. Иерархия компонентов ... 117
5.2. Create React App ... 117
5.2.1. Создание приложения React с помощью Create React App ... 117
5.3. Состояние и подключаемые обработчики React Hooks ... 124
5.4. Упражнения ... 128
Итоги ... 128

Глава 6. Клиент GraphQL 130
6.1. Apollo Client 131
6.1.1. Добавление Apollo Client в приложение React 131
6.1.2. Обработчики Apollo Client 134
6.1.3. Переменные GraphQL 138
6.1.4. Фрагменты GraphQL 139
6.1.5. Кеширование в Apollo Client 141
6.2. Мутации GraphQL 143
6.2.1. Создание узлов 143
6.2.2. Создание отношений 145
6.2.3. Изменение и удаление 146
6.3. Управление состоянием клиента с помощью GraphQL 147
6.3.1. Локальные поля и реактивные переменные 147
6.4. Упражнения 151
Итоги 152

Часть III. Задачи разработки полного цикла 153

Глава 7. Добавление авторизации и аутентификации 154
7.1. Авторизация в GraphQL: простейший подход 155
7.2. Веб-токены JSON Web Token 158
7.3. Директива схемы @auth 162
7.3.1. Правила и операции 163
7.3.2. Правило авторизации isAuthenticated 164
7.3.3. Правило авторизации roles 165
7.3.4 Правило авторизации allow 168
7.3.5. Правило авторизации where 169
7.3.6. Правило авторизации bind 171
7.4. Auth0: JWT как услуга 172
7.4.1. Настройка Auth0 172
7.4.2. Auth0 React 175
7.5. Упражнения 186
Итоги 186

Глава 8. Развертывание приложения GraphQL 187
8.1. Развертывание приложения GraphQL 187
8.1.1. Преимущества развертывания в бессерверном окружении 188
8.1.2. Недостатки развертывания в бессерверном окружении 189
8.1.3. Обзор подхода к развертыванию приложения GraphQL в бессерверном окружении 189
8.2. База данных Neo4j Aura как услуга 190
8.2.1. Создание кластера Neo4j Aura 191
8.2.2. Подключение к кластеру Neo4j Aura 193

8.2.3. Выгрузка данных в Neo4j Aura ... 196
8.2.4. Исследование графа с помощью Neo4j Bloom ... 198
8.3. Развертывание приложения React с помощью Netlify Build ... 201
8.3.1. Добавление сайта в Netlify ... 202
8.3.2. Настройка переменных окружения для сборок Netlify ... 210
8.3.3. Предварительное развертывание в Netlify ... 213
8.4. Развертывание GraphQL в виде бессерверной функции с помощью AWS Lambda и Netlify Functions ... 216
8.4.1. GraphQL API в виде бессерверной функции ... 216
8.4.2. dev: интерфейс командной строки Netlify ... 218
8.4.3. Преобразование GraphQL API в функцию Netlify ... 219
8.4.4. Добавление собственного домена в Netlify ... 222
8.5. Наш подход к развертыванию ... 224
8.6. Упражнения ... 225
Итоги ... 225

Глава 9. Продвинутые возможности GraphQL ... 227

9.1. Абстрактные типы GraphQL ... 227
9.1.1. Интерфейсы ... 228
9.1.2. Объединения ... 229
9.1.3. Использование абстрактных типов с библиотекой Neo4j GraphQL ... 230
9.2. Разбиение на страницы с помощью GraphQL ... 242
9.2.1. Разбиение на страницы по смещению ... 243
9.2.2. Разбиение на страницы с помощью курсора ... 244
9.3. Свойства отношений ... 248
9.3.1. Интерфейсы и директива @relationship ... 249
9.3.2. Создание свойств отношений ... 250
9.4. В заключение ... 251
9.5. Упражнения ... 252
Итоги ... 253

Предметный указатель ... 254

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications Co. очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую предоставлять вам качественные материалы.

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!

Благодарности

Работа над книгой – длительный процесс, требующий помощи и поддержки многих людей. Невозможно отметить всех, кто помог в создании этой книги, не упустив никого. Конечно, эта книга была бы невозможна без всех участвовавших в создании удивительных технологий, о которых мы рассказываем.

Спасибо Майклу Стивенсу (Michael Stephens) за предложение написать книгу о GraphQL и помощь в развитии идеи полного стека GraphQL, Карен Миллер (Karen Miller) за рецензирование ранних версий каждой главы, а также всем сотрудникам издательства Manning, принявшим участие в проекте: Дугу (Doug), Александару (Aleksandar), Энди (Andy), Кристиану (Christian), Мелоди (Melody), Ниеку (Niek), Гордану (Gordan) и Марии (Marija). Спасибо моей семье, что терпели меня во время работы над этой книгой. Особое спасибо сообществу за помощь в проверке идей, изложенных в этой книге, а также за подробные отзывы и вклад в развитие библиотеки Neo4j GraphQL.

Дорогие мои рецензенты: Андрес Сакко (Andres Sacco), Брендон Фрайер (Brandon Friar), Кристофер Хаупт (Christopher Haupt), Дамиан Эстебан (Damian Esteban), Данило Зекович (Danilo Zekovic), Дениз Вехби (Deniz Vehbi), Ферит Топку (Ferit Topcu), Франс Оилинки (Frans Oilinki), Густаво Гомес (Gustavo Gomes), Харш Раваль (Harsh Raval), Иво Санчес Чека Кросато (Ivo Sánchez Checa Crosato), Хосе Антонио Эрнандес Ороско (Jose Antonio Hernandez Orozco), Хосе Сан Леандро (Jose San Leandro), Кевин Реди (Kevin Ready), Константинос Леймонис (Konstantinos Leimonis), Кшиштоф Камычек (Krzysztof Kamyczek), Микеле Аддуци (Michele Adduci), Мигель Исидоро (Miguel Isidoro), Ричард Мейнсен (Richard Meinsen), Ричард Воган (Richard Vaughan), Роб Лейси (Rob Lacey), Рональд Борман (Ronald Borman), Райан Хубер (Ryan Huber), Сейт Кумар Саху (Satej Kumar Sahu), Симеон Лейзерзон (Simeon Leyzerzon), Стефан Туральски (Stefan Turalski), Таня Вилке (Tanya Wilke), Теофанис Деспоудис (Theofanis Despoudis) и Владимир Пасман (Vladimir Pasman), спасибо вам! Ваши предложения помогли сделать эту книгу лучше.

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 запи-

си 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 со-

ответствуют 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 запросов). 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

запросе. 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 применять 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 применяется 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",
  neo4j.auth.basic("neo4j", "letmein"));
                                        ← Создание экземпляра драйвера с указанием строки подключения к базе данных
                                        ← 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")
  .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( () => {
    session.close();
  )
                                        ← Запуск запроса в транзакции с автоматическим подтверждением; он возвращает объект Promise
                                        ← Не забывайте закрывать сеансы!
```

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 создавать 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.

Глава 2

Графовое мышление с GraphQL

В этой главе:

- описание требований к приложению обзора компаний;
- преобразование требований в определения типов GraphQL;
- реализация функций разрешения для выборки данных на основе определений типов с применением прямолинейного подхода;
- использование Apollo Server для объединения определений типов с функциями разрешения и обслуживания конечной точки GraphQL;
- выполнение запросов к конечной точке GraphQL с помощью Apollo Studio.

В этой главе мы спроектируем GraphQL API для приложения обзора компаний. Сначала определим требования к этому приложению, затем опишем GraphQL API, отвечающий этим требованиям. Потом исследуем подходы к реализации логики выборки данных для этого API. И наконец, изучим приемы объединения наших определений типов GraphQL и функций разрешения для обслуживания GraphQL API, задействовав Apollo Server и Apollo Studio для выполнения запросов к нему. При создании API часто полезно досконально разобраться в предметной области и общих шаблонах доступа – другими словами, какие задачи должен решать API? Подход к разработке, основанный на GraphQL, позволяет создать API, сначала рассматривая данные и определяя схему GraphQL, описывающую эти данные, которая затем служит планом для реализации API.

2.1. Данные приложения – это граф

Граф – это фундаментальная структура данных, состоящая из узлов (сущностей или объектов) и связей, соединяющих узлы. Граф – это интуитивно понятная модель, которую можно использовать для представления данных во множестве различных областей. Часто, упражняясь в создании моделей данных и изучая бизнес-требования, мы в конечном итоге рисуем диаграмму из объектов и стрелок, показывающих связи между ними. Это граф!

Разработка на основе GraphQL

Парадигма разработки на основе GraphQL – это подход к созданию приложений GraphQL API. Процесс начинается с описания определений типов, синтезированных из бизнес-требований. Затем эти определения становятся основой для реализации API – программного кода, выбирающего данные из базы данных и реализующего отображение этих данных в пользовательском интерфейсе. Разработка на основе GraphQL – это мощный подход, позволяющий одновременно реализовывать серверную и клиентскую части системы после определения типов GraphQL.

Давайте пройдем этот процесс, начав разработку нашего приложения обзора компаний. Вот требования к нашему приложению:

1. Как пользователь я хочу иметь возможность получить список компаний по категориям, местонахождению и названию.
2. Как пользователь я хочу иметь возможность просматривать сведения о каждой компании (название, описание, адрес, фотографии и т. д.).
3. Как пользователь я хочу иметь возможность просматривать отзывы о каждой компании, включая сводные данные по каждому направлению, и ранжировать результаты поиска по оценкам, оставленным другими пользователями.
4. Как пользователь я хочу иметь возможность создать свой отзыв о компании.
5. Как пользователь я хочу иметь возможность связать своих друзей и пользователей, чьи предпочтения совпадают с моими, чтобы следить за отзывами своих друзей.
6. Как пользователь я хочу иметь возможность получать персональные рекомендации на основе ранее написанных отзывов и моей социальной сети.

Теперь, определив требования к приложению, подумаем о требованиях к данным и описывающей их модели.

Во-первых, что такое сущности. Они будут служить узлами в нашем графе. Сущностями могут быть пользователи, компании, отзывы и фотографии (см. рис. 2.1).

Рис. 2.1. Сущности будут служить узлами

Далее, как эти сущности связаны. Связи моделируются как отношения между сущностями, и то, что мы описали, представляет собой граф (рис. 2.2). Добавим следующие отношения:

1. Пользователи пишут отзывы.
2. Отзывы связаны с компаниями.

3. Пользователи выгружают фотографии.
4. Фотографии помечены как компании.

Теперь, описав требования к данным в виде графа, можно начать думать о том, как построить GraphQL API для работы с этим графом данных.

Рис. 2.2. Добавление отношений, связывающих узлы графа

2.2. Графы в GraphQL

GraphQL моделирует предметную область в виде графа. Работая над приложением GraphQL, мы определяем эту модель, создавая схему GraphQL, для чего записываем определения типов GraphQL. В схеме мы определяем типы узлов, поля, доступные в каждом узле, и взаимосвязи между узлами. Наиболее распространенный способ создания схемы GraphQL – использование языка определения схем (Schema Definition Language, SDL) GraphQL. В этом разделе, исходя из требований к приложению, мы создадим схему GraphQL, моделирующую предметную область приложения обзора компаний с применением определений типов GraphQL.

2.2.1. *Моделирование API с применением определений типов: разработка на основе GraphQL*

Преобразовав бизнес-требования в необходимую графовую модель данных, мы теперь можем составить формальные определения типов GraphQL на языке определения схем. С помощью GraphQL SDL мы определим типы, поля в каждом типе и взаимосвязи между типами. Описание данных на языке GraphQL SDL – это еще один способ представления графовой модели данных, которую мы описали в предыдущем разделе. Наши определения типов станут спецификацией для API и будут определять остальную часть нашей реализации. Этот процесс известен как *разработка на основе GraphQL*.

Службы GraphQL могут быть реализованы на любом языке, поэтому синтаксис какого-то определенного языка программирования подходит не для всех реализаций GraphQL. По этой причине для определения типов GraphQL используется GraphQL SDL – собственный язык определения схем, не зависящий от языка программирования. В главе 1 мы видели образцы синтаксиса языка определения схем GraphQL на примере простой схемы GraphQL с фильмами и актерами. Теперь, вооруженные этими знаниями, создадим определения типов GraphQL для нашего приложения обзора компаний, отталкиваясь от требований, описанных в предыдущем разделе (листинг 2.1).

Другие способы представления типов GraphQL

SDL – не единственный способ создания определений типов. Каждая реализация GraphQL (например, graphql.js, эталонная реализация, используемая в большинстве проектов Node.js JavaScript GraphQL) может предоставлять свой программный API для определения типов GraphQL. Фактически при разборе SDL создается именно это представление объекта для работы со схемой GraphQL. Этот подход к созданию типов GraphQL тоже может использоваться разработчиком API и часто является лучшим вариантом при программном определении типов GraphQL, например при создании типов из существующих классов.

Листинг 2.1. Определения типов GraphQL для приложения обзора компаний

```
type Business {
  businessId: ID!
  name: String
  address: String
  avgStars: Float
  photos: [Photo!]!
  reviews: [Review!]!
}

type User {
  userId: ID!
  name: String
  photos: [Photo!]!
  reviews: [Review!]!
}

type Photo {
  business: Business!
  user: User!
  photoId: ID!
  url: String
}

type Review {
  reviewId: ID!
  stars: Float
  text: String
  user: User!
  business: Business!
}
```

Каждый тип объекта или сущности в графе становится типом GraphQL

Каждый тип должен иметь некоторое поле, однозначно идентифицирующее этот объект

Поля могут быть ссылками на другие типы – в данном случае описывается отношение «один ко многим»

Ссылки на ссылки могут также представлять отношения «один к одному»

Обратите внимание, что определенные нами сущности стали типами GraphQL, свойства сущностей – полями типов, а связи, или отношения, соединяющие типы, определяются как поля, ссылающиеся на другие типы. Каждый тип содержит поля, которые могут быть скалярными типами, объектами или списками.

Каждый тип должен иметь некоторое поле, однозначно идентифицирующее конкретный объект. `ID` – это специальный скаляр в GraphQL, используемый для представления этого идентифицирующего поля. Внутренне поля `ID` обрабатываются как строки. Знак восклицания (`!`) указывает, что поле является обязательным, т. е. в нашем GraphQL API не может быть объекта `User` с пустым значением в поле `userId`. Квадратные скобки `[]` указывают, что это отношение «один ко многим»; один `User` (пользователь) может создать ноль или более обзоров, а `Review` (отзыв) может быть написан только одним пользователем. Для описания отношения «один к одному» достаточно просто опустить квадратные скобки, сообщив, что это поле не является массивом.

Встроенные типы GraphQL

Ниже перечислены встроенные типы, поддерживаемые языком схем GraphQL:

- `String`;
- `Int`;
- `Float`;
- `Boolean`;
- `ID`.

По умолчанию все типы поддерживают значение `null`, т. е. `null` является допустимым значением для поля. Используйте восклицательный знак (!), чтобы указать, что поле не может иметь значение `null`. Например, `Int!` – это непустое целое число.

Чтобы определить поле-список, используйте квадратные скобки `[]`. Например, `[Int]` – это список целых чисел. Скобки и восклицательные знаки можно комбинировать. Например, `[String!]` – это список непустых строк: каждый элемент в списке должен иметь значение `String`, но сам список может быть пустым, а `[String]!` – это непустой список строк, который может содержать значения `null`.

После определения типов нужно определить точки входа в наш API. Точки входа для операций чтения определяются в специальном типе `Query`. Точки входа для операций записи определяются в специальном типе `Mutation`. В этой главе мы сосредоточимся только на типе `Query`, а тип `Mutation` рассмотрим в главе 4, когда начнем обновлять данные в базе данных. В дополнение к типам `Query` и `Mutation` существует третий специальный тип, определяющий точки входа, – `Subscription`. Этот тип представляет функции публикации событий GraphQL, но мы не будем обсуждать их в этой книге.

Точки входа в наш API должны соответствовать требованиям приложения. Проще говоря, спросите себя: «Какие операции должен выполнить клиент?» Эти требования должны определять, какие поля в типах `Query` и `Mutation` следует определить. Для начала сосредоточимся только на требованиях к чтению данных (листинг 2.2).

Листинг 2.2. Поля в типе Query как точки входа в API

```
type Query {
  allBusinesses: [Business!]!
  businessBySearchTerm(search: String!): [Business!]!
  userById(id: ID!): User
}
```

Покончив с определениями типов GraphQL, можно попробовать создать некоторые запросы для нашего приложения. Рассмотрим для примера запрос, заполняющий страницу результатами поиска по строке, введенной пользователем (листинг 2.3).

Листинг 2.3. Запрос для поиска компаний и отзывов

```
{
  businessBySearchTerm(search: "Library") {
    name
    avgStars
    reviews {
      stars
      text
      user {
        name
      }
    }
  }
}
```

С помощью этого запроса можно искать компании, названия которых включают слово «Library» (библиотеки), и просматривать информацию о них, а также отзывы о компаниях и сведения о пользователях, оставивших их.

Это здорово, но есть несколько проблем. Что произойдет, если обнаружится слишком много совпадений с запросом «Library»? Что, если для какой-то компании пользователи оставили тысячи отзывов? Клиентское приложение будет перегружено данными. Кроме того, мы едва ли захотим показывать результаты в произвольном порядке; они должны, например, упорядочиваться по названиям компаний в порядке возрастания или убывания.

Аргументы и поля

Важно понимать различие между аргументами и полями. Например, `first` и `offset` – это аргументы, а `name` и `address` – поля. Аргументы определяются в круглых скобках после имени поля и передаются функциям разрешения. Поля определяются в фигурных скобках после имени объекта и представляют атрибуты этого объекта. Поля можно рассматривать как хранилища значений, а аргументы используются скорее как селекторы и передаются в операциях GraphQL.

Добавление в API постраничного просмотра и упорядочения

GraphQL не имеет встроенной семантики для фильтрации, разбиения на страницы или упорядочения; разработчик API должен сам добавить ее в схему GraphQL, если сочтет это необходимым и актуальным.

Для поддержки постраничного просмотра результатов добавим в наш API аргумент `first` (интерпретируйте его как ограничение), чтобы клиент мог указать ко-

личество возвращаемых объектов. Мы добавим это ограничение и для корневого поля запроса, и для любых полей отношений, описывающих отношения «один ко многим». Также добавим аргумент `offset` (чтобы реализовать пропуск нескольких первых результатов), определяющий количество записей, которые нужно пропустить перед возвратом результатов. Это позволит клиенту реализовать разбиение результатов на страницы.

Листинг 2.4. Обновленные определения типов `Query` и `Business` с дополнительными аргументами `first` и `offset`

```
type Business {
  businessId: ID!
  name: String
  address: String
  avgStars: Float
  photos(first: Int = 3, offset: Int = 0): [Photo!]!
  reviews(first: Int = 3, offset: Int = 0): [Review!]!
}

type Query {
  allBusinesses(first: Int = 10, offset: Int = 0): [Business!]!
  businessBySearchTerm(
    search: String
    first: Int = 10
    offset: Int = 0
  ): [Business]
  userById(id: ID!): User
}
```

Здесь мы добавляем аргументы `first` и `offset` в поле `reviews` в типе `Business`. Это означает, что теперь есть возможность управлять разбиением на страницы с вложенными объектами `Review` для каждой компании, возвращаемой запросом

Здесь мы добавляем аргументы `first` и `offset` в поле `allBusinesses`, позволяя клиенту задать, сколько предприятий пропустить и сколько вернуть. Обратите внимание, что мы присваиваем значения по умолчанию – если они не будут указаны явно, то запрос получит значения по умолчанию: 10 в аргументе `first` и 0 в аргументе `offset`, – и вернет первые 10 результатов

В поле `userById` можно не добавлять аргументы `first` и `offse`, потому что оно гарантированно возвращает не более одного результата

Параметры разбиения на страницы

В GraphQL есть несколько шаблонов реализации разбиения на страницы. Здесь мы сосредоточимся на одном из самых простых – «начало/смещение». В числе других вариантов можно назвать шаблон нумерации страниц и разбиение на страницы с помощью курсора, например Relay Cursor Connections. Разбиение с помощью курсора Relay Cursor Connection рассматривается в главе 9.

Эти определения решают задачу разбивки на страницы, но нам осталось еще решить проблему упорядочения. Это необходимо для отображения результатов поиска, чтобы пользователь мог просматривать компании в определенном порядке. Для этого добавим перечисление, определяющее варианты упорядочения полей типа `[Business]` в нашем GraphQL API.

Листинг 2.5. Варианты упорядочения полей типа [Business]

```
enum BusinessOrdering {
  name_asc
  name_desc
}
```

enum – это встроенный тип GraphQL, имеющий ограниченный набор допустимых значений

К каждому полю, для которого требуется поддерживать определенный порядок, добавляются два параметра: один (с суффиксом _asc) определяет упорядочение по возрастанию, а другой (с суффиксом _desc) – по убыванию

Обычно имена членов перечислений записываются символами верхнего регистра (например, NAME_ASC); однако в нашем случае члены перечисления описывают имена полей, поэтому мы сделаем исключение и запишем их имена так же, как имена полей любых других типов. Теперь, определив перечисление, добавим его поле как дополнительный аргумент в поле типа Query, определяющего запрос для поиска компаний (листинг 2.6).

Листинг 2.6. Добавление упорядочения компаний в возвращаемом наборе результатов

```
type Query {
  allBusinesses(first: Int = 10, offset: Int = 0): [Business!]!
  businessBySearchTerm(
    search: String!
    first: Int = 10
    offset: Int = 0
    orderBy: BusinessOrdering = name_asc
  ): [Business!]!
  userById(id: ID!): User
}
```

Для поля businessBySearchTerm добавлен аргумент orderBy типа BusinessOrdering

Теперь мы можем использовать новые аргументы, управляющие упорядочением разбивания на страницы. В листинге 2.7 показан обновленный запрос поиска компаний с названиями, включающими слово «Library», который возвращает пять компаний с самым высоким рейтингом и по два отзыва для каждой из них.

Листинг 2.7. Запрос GraphQL для поиска компаний и соответствующих им отзывов

```
{
  businessBySearchTerm(search: "Library", first: 5, orderBy: name_desc) {
    name
    avgStars
    reviews(first: 2) {
      stars
      text
      user {
        name
      }
    }
  }
}
```

Обычно при использовании запросов с аргументами бывает желательно передавать эти аргументы через переменные, которые можно изменять в процессе работы, поэтому в приложениях принято не конструировать запросы «на лету», а использовать так называемые параметризованные запросы и объекты со значениями переменных. Чтобы добиться желаемого в GraphQL, сначала нужно объявить переменные, представляющие аргументы, указать их типы, а затем включить эти переменные в запрос, добавив к их именам префикс $. В листинге 2.8 показано, как будет выглядеть наш запрос с использованием переменных GraphQL.

Листинг 2.8. Запрос GraphQL для поиска компаний и соответствующих им отзывов с поддержкой разбиения на страницы

```
query businessSearch(
  $searchTerm: String!
  $businessLimit: Int
  $businessSkip: Int
  $businessOrder: BusinessOrdering
  $reviewLimit: Int
) {
  businessBySearchTerm(
    search: $searchTerm
    first: $businessLimit
    offset: $businessSkip
    orderBy: $businessOrder
  ) {
    name
    avgStars
    reviews(first: $reviewLimit) {
      stars
      text
      user {
        name
      }
    }
  }
}
```

Обратите внимание, что теперь запрос включает дополнительную информацию наряду с объявлениями переменных. Мы явно указываем *тип и имя операции* GraphQL. Тип операции – `query`, `mutation` или `subscription`. Раньше мы использовали сокращенную форму записи, исключая тип операции, в отсутствие которого по умолчанию подразумевался тип операции `query`. Тип `mutation` мы рассмотрим позже в этой книге. Тип операции можно не указывать, если не задано имя операции, отсутствуют определения переменных и операция имеет тип `query`, подразумеваемый по умолчанию.

Другая дополнительная информация здесь – имя операции, в данном случае `businessSearch`. Имя явно идентифицирует операцию и может пригодиться во вре-

мя отладки и для нужд журналирования, потому что искать в журнале запросы по имени операции намного проще. Вместе с запросом GraphQL мы также передаем объект, содержащий значения переменных:

```
{
  searchTerm: "Library",
  businessLimit: 5,
  businessOrder: "name_desc",
  reviewLimit: 2
}
```

На данный момент мы не можем отправить запрос к API, которого пока не существует, поэтому исправим эту проблему и реализуем несколько функций разрешения для выборки данных!

2.2.2. Выборка данных с помощью функций разрешения

В соответствии с подходом к разработке, основанным на GraphQL, следующим нашим шагом будет реализация фактического извлечения данных из хранилища. Для этого мы напишем функции, называемые *функциями разрешения* (resolver function), которые содержат логику извлечения данных из хранилища. Функции разрешения – это автономные функции, предназначенные для получения данных для одного поля типа GraphQL. Их можно рассматривать как основную единицу выполнения в службе GraphQL. Функции разрешения вызываются каскадно, начиная с корневой функции (поле в запросе типа `query`, `mutation` или `subscription`), пока не будут разрешены все запрошенные поля. Данные, извлеченные предыдущей функцией разрешения, передаются вложенным функциям через параметр `obj`.

Функции разрешения можно рассматривать как методы, сопровождающие определения типов GraphQL и фактически делающие схему GraphQL выполняемой. Схема GraphQL должна иметь функции разрешения для всех полей (взамен функций, которые не определены явно, используется функция разрешения по умолчанию), поэтому набор функций соответствует определениям типов и называется картой разрешения.

Сигнатура функции разрешения

Каждая функция разрешения принимает четыре аргумента:

- `obj` – ранее разрешенный объект; не используется в корневой функции;
- `args` – аргументы поля, используемого в запросе GraphQL;
- `context` – объект с контекстными данными, такими как информация об авторизации или соединение с базой данных;
- `info` – объект `GraphQLResolveInfo` с версией запроса, а также полной схемой GraphQL и другими метаданными о запросе и схеме.

Функции разрешения возвращают следующие данные в зависимости от определения типа разрешаемого поля:

- скалярное или объектное значение;
- массив;

- объект `Promise`, представляющий результат, который будет готов в какой-то момент в будущем;
- `undefined` или `null`.

Функции разрешения по умолчанию

Если для какого-то поля в запросе не указана функция разрешения, то будет вызвана функция по умолчанию, которая получит уже разрешенные данные (аргумент `obj`, упомянутый выше). Функция по умолчанию вернет свойство из параметра `obj` с именем поля. Например, функция по умолчанию для поля `name` в типе `Business` будет выглядеть примерно так:

```
Business: {
  name: (obj, args, context, info) => {
    return obj.name
  }
}
```

2.2.3. *Наша первая функция разрешения*

Давайте реализуем функции разрешения для созданных нами определений типов (листинг 2.9). Первым делом нужно определить, какие данные должны возвращаться, поэтому создадим некоторый статический набор данных, имитирующий хранилище. Мы просто создадим несколько объектов и сохраним их в объекте с именем `db`, который будем рассматривать как макет базы данных. Внедрим этот объект `db` с фиктивными данными в объект `context`, чтобы он был доступен каждой функции разрешения.

Листинг 2.9. Примеры данных о компаниях, отзывов и пользователях, находящихся в фиктивном хранилище

```
const businesses = [
  {
    businessId: "b1",
    name: "Missoula Public Library",
    address: "301 E Main St, Missoula, MT 59802",
    reviewIds: ["r1", "r2"],
  },
  {
    businessId: "b2",
    name: "San Mateo Public Library",
    address: "55 W 3rd Ave, San Mateo, CA 94402",
    reviewIds: ["r3"],
  },
];

const reviews = [
  {
    reviewId: "r1",
    stars: 3,
```

```
    text: "Friendly staff. Interlibrary loan is super fast",
    businessId: "b1",
    userId: "u1",
  },
  {
    reviewId: "r2",
    stars: 4,
    text: "Easy downtown access, lots of free parking",
    businessId: "b1",
    userId: "u2",
  },
  {
    reviewId: "r3",
    stars: 5,
    text: "Lots of glass and sunlight for reading.",
    businessId: "b1",
    userId: "u1",
  },
];

const users = [
  {
    userId: "u1",
    name: "Will",
    reviewIds: ["r1", "r2"],
  },
  {
    userId: "u2",
    name: "Bob",
    reviewIds: ["r3"],
  },
];

const db = { businesses, reviews, users };
```

Предположим, что эти объекты передаются функциям разрешения в объекте `context`, как при передаче объекта подключения к базе данных.

Имитация данных в GraphQL

Вместо статического объекта, имитирующего базу данных, можно также использовать поддержку имитации данных сервера Apollo и с ее помощью создать функции разрешения, которые возвращают фиктивные данные. Эту возможность удобно использовать для нужд тестирования, а также для обеспечения одновременной работы групп разработчиков, занимающихся созданием серверной части и пользовательского интерфейса. Мы можем быть уверены в релевантности данных, возвращаемых этой поддержкой, потому что она использует механизм интроспекции схемы и систему типов GraphQL, гарантируя соответствие формы имитируемых данных нашим определениям типов GraphQL. Узнать больше об имитации данных с помощью Apollo Server можно в документации: **http://mng.bz/Pnlw**.

Исходя из наших определений типов GraphQL, исходная карта разрешения будет выглядеть, как показано в листинге 2.10.

Листинг 2.10. Исходная карта разрешения

```
const resolvers = {
  Query: {
    allBusinesses: (obj, args, context, info) => {
      // TODO: вернуть список всех компаний
    },
    businessBySearchTerm: (obj, args, context, info) => {
      // TODO: отыскать компании, соответствующие условиям в запросе
    }
  },
  Business: {
    reviews: (obj, args, context, info) => {
      // TODO: отыскать отзывы для конкретной компании
    },
    avgStars: (obj, args, context, info) => {
      // TODO: вычислить средний рейтинг
    }
  },
  Review: {
    user: (obj, args, context, info) => {
      // TODO: отыскать пользователя, оставившего отзыв
    },
    business: (obj, args, context, info) => {
      // TODO: отыскать компанию для этого отзыва
    }
  },
  User: {
    reviews: (obj, args, context, info) => {
      // TODO: отыскать все отзывы, оставленные этим пользователем
    }
  }
};
```

Обратите внимание, что нам не нужно беспокоиться о реализации тривиальных функций разрешения, таких как `Business.name`, которые с успехом могут заменить функции разрешения по умолчанию. Начнем с реализации функции `allBusiness` (листинг 2.11). Эта функция просто извлекает все компании из хранилища и возвращает их, не заботясь о разбивке на страницы или упорядочении. Помните, что в этом примере наше хранилище представляет вложенный объект, передаваемый в функции через объект `context`. (Как на самом деле внедрить этот объект, мы увидим в следующем разделе.)

Листинг 2.11. Корневая функция разрешения: allBusinesses

```
Query: {
    allBusinesses: (obj, args, context, info) => {
      return context.db.businesses;
    }
}
```

Мы разрешаем поле в типе `Query`, поэтому эта функция определяется внутри определения типа `Query`

Здесь мы видим стандартную сигнатуру функции разрешения. В аргументе `obj` в этом случае будет передано значение `null`, так как это корневая функция разрешения и пока нет данных для обработки. В `args` тоже будет передано значение `null`, так как это поле не имеет никаких аргументов. Аргумент `context`, однако, будет содержать наш статический объект данных

Возвращается массив компаний, перечисленных в объекте `db`, доступ к которому осуществляется через объект `context`

Теперь, реализовав нашу первую функцию разрешения, посмотрим, как объединить определения типов GraphQL и функцию разрешения для обслуживания GraphQL API с помощью сервера Apollo.

2.3. Объединение определений типов и функций разрешения в Apollo Server

Итак, мы определили типы GraphQL и создали первую функцию разрешения для получения данных из хранилища. Теперь пришло время собрать их вместе и запустить сервер GraphQL с Apollo Server. Apollo Server доступен в виде пакета npm, поэтому установим его с помощью `npm`:

```
npm install apollo-server graphql
```

2.3.1. *Apollo Server*

В листинге 2.12 приводится содержимое файла index.js, который будет использовать наши определения типов и функции разрешения, а также Apollo Server для обслуживания GraphQL API на основе этих определений.

Листинг 2.12. Файл index.js сервера GraphQL, использующий Apollo Server

```
const ApolloServer = require('apollo-server');

const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: { db }
});

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

Импорт ApolloServer из только что установленного пакета

Создание экземпляра сервера

Определения наших типов

Наши функции разрешения

db – это наш фиктивный объект данных, внедренный в контекст. Этот объект будет доступен в каждой функции разрешения

Запуск сервера и начало обработки входящих запросов GraphQL

2.3.2. *Apollo Studio*

По умолчанию Apollo Server обслуживает конечную точку GraphQL, предназначенную для приема запросов `POST`, а запросы `GET`, отправляемые из веб-браузера

по тому же URL (в нашем случае **http://localhost:4000**), Apollo Server будет передавать расширению Apollo Studio, установленному в браузере (рис. 2.3).

Рис. 2.3. Обработка запросов в Apollo Studio

Apollo Studio можно использовать для просмотра определений типов и схемы GraphQL API, а также для выполнения запросов, передачи изменений и просмотра результатов. На данный момент мы реализовали только одну функцию разрешения – `allBusinesses`. Давайте проверим ее, выполнив следующий запрос в Apollo Studio:

```
{
  allBusinesses {
    name
  }
}
```

Для обработки этого запроса будет вызвана функция разрешения `allBusinesses`, которая вернет объект `businesses` из нашего фиктивного хранилища. Затем, поскольку запрашивается только поле `name` объектов типа `Business`, для возврата названий всех компаний будет использоваться функция разрешения по умолчанию (рис. 2.4).

Рис. 2.4. Результат выполнения простого запроса в Apollo Studio

Поэкспериментировав с запросом в Apollo Studio, вы быстро осознаете необходимость реализации остальных функций разрешения. Вернемся к нашей карте разрешения и сделаем это.

2.3.3. *Реализация функций разрешения*

Мы определили некоторые фиктивные данные и написали первую функцию разрешения `allBusinesses`, которая просто возвращает названия всех компаний в нашей фиктивной базе данных. Теперь попробуем реализовать более сложные функции разрешения, такие как `businessBySearchTerm`, позволяющие фильтровать результаты по условиям в запросе пользователя, и функции разрешения массивов, такие как `Business.reviews`, обрабатывающие связи между компаниями и отзывами.

Корневая функция разрешения: businessBySearchTerm

Корневыми называют функции разрешения, соответствующие точкам входа в API. В нашем определении типа `Query` имеются следующие точки входа:

```
type Query {
  allBusinesses: [Business!]!
  businessBySearchTerm(
    search: String!
    first: Int = 10
    offset: Int = 0
    orderBy: BusinessOrdering = name_asc
  ): [Business!]!
  userById(id: ID!): User
}

enum BusinessOrdering {
  name_asc
  name_desc
}
```

Мы уже реализовали корневую функцию разрешения `allBusinesses` в предыдущем разделе. Тот пример был довольно простым благодаря отсутствию аргументов. Теперь нам предстоит реализовать функцию `businessesBySearchTerm`, которая принимает искомую строку, а также аргументы, определяющие порядок сортировки и разбиение на страницы, как показано в листинге 2.13.

Листинг 2.13. Корневая функция разрешения: businessBySearchTerm

```
businessBySearchTerm: (obj, args, context, info) => {
  const compare = (a, b) => {
    const [orderField, order] = args.orderBy.split("_");
    const left = a[orderField],
       right = b[orderField];
    if (left < right) {
      return order === "asc" ? -1 : 1;
    } else if (left > right) {
      return order === "desc" ? -1 : 1;
    } else {
      return 0;
    }
  };
  return context.db.businesses
    .filter(v => {
      return v["name"].indexOf(args.search) !== -1;
    })
    .slice(args.offset, args.first)
    .sort(compare);
}
```

Это корневая функция разрешения, поэтому параметр `obj` содержит `null`, но в объекте `args` функция получит аргументы запроса GraphQL – в данном случае `orderBy`, `search`, `first` и `offset`. В наших определениях типов для `orderBy`, `first` и `offset` указаны значения по умолчанию, а `search` является обязательным полем, поэтому можно быть уверенными, что все эти аргументы будут иметь значения

Здесь определяется функция сравнения, выполняющая сортировку с использованием нашего перечисления `BusinessOrdering`. Мы разбиваем значение `orderBy` по символу подчеркивания, чтобы определить имя поля и направление сортировки (например, `name_asc` означает сортировку по полю name в порядке возрастания (`asc`))

Фильтрация названий компаний: свойство name должно содержать искомую строку, переданную в запросе GraphQL

Функция `slice` используется для разбиения на страницы в соответствии с аргументами `first`/`offset`

Применение функции сравнения compare для сортировки результатов в соответствии со значением, указанным в аргументе `orderBy`. Если аргумент `orderBy` не указан в пользовательском запросе, будет использоваться значение по умолчанию `name_asc`, заданное в определении типа

Функция разрешения массива: Business.reviews

Наши предыдущие корневые функции разрешения возвращали массивы объектов, но точно так же можно возвращать массивы объектов из функций разрешения для некорневых функций разрешения, если поле определено как список (как, например, поле `Business.reviews`, имеющее тип `[Review]`, – список объектов `Review`). В функциях разрешения некорневого уровня параметр `obj` будет содержать все ранее разрешенные данные. Например, если мы сначала выполним `Query.businessBySearchTerm` для выборки компаний, то результаты этой функции будут переданы следующей за ней функции `Business.reviews`. Воспользуемся этими данными и реализуем функцию `Business.reviews` (листинг 2.14).

Листинг 2.14. Корневая функция разрешения

```
Business: {
  reviews: (obj, args, context, info) => {
```

```
    return obj.reviewIds.map(v => {
      return context.db.reviews.find(review => {
        return review.reviewId === v;
      });
    });
  },
}
```

Скалярная функция разрешения: Business.avgStars

Выше уже упоминались функции разрешения по умолчанию, просто возвращающие свойство объекта с тем же именем, что и поле в параметре `obj`, но иногда нужны функции разрешения, возвращающие скалярные значения в обход функции разрешения по умолчанию. Хорошим примером могут служить агрегированные значения. Поле `Business.avgStars` – это агрегированное поле. Чтобы вернуть его, нужно отыскать все отзывы для определенной компании, а затем вычислить среднее количество звезд по всем этим отзывам и вернуть одно скалярное значение (листинг 2.15).

Листинг 2.15. Скалярная функция разрешения

```
avgStars: (obj, args, context, info) => {
  const reviews = obj.reviewIds.map(v => {
    return context.db.reviews.find(review => {
      return review.reviewId === v;
    });
  });

  return (
    reviews.reduce((acc, review) => {
      return acc + review.stars;
    }, 0) / reviews.length
  );
}
```

Объектная функция разрешения: Review.user

До сих пор мы видели функции, возвращающие скалярные значения и массивы; теперь функцию, возвращающую один объект (листинг 2.16). В наших определениях тип `Review` связан с единственным объектом `User`, т. е. `Review.user` – это поле, содержащее объект, а не список.

Листинг 2.16. Объектная функция разрешения

```
Review: {
  user: (obj, args, context, info) => {
    return context.db.users.find(user => {
      return user.userId === obj.userId;
    });
```

```
  }
}
```

Закончив реализацию этой последней функции разрешения, мы можем продолжить исследование запросов к нашему GraphQL API, выполняемых с помощью Apollo Studio.

2.3.4. Выполнение запросов с помощью Apollo Studio

Теперь, реализовав все необходимые функции разрешения, вернемся в Apollo Studio, открыв страницу **http://localhost:4000/** в веб-браузере. Для начала отыщем все компании, названия которых включают слово «Library» (рис. 2.5).

```
{
  businessBySearchTerm(search: "Library") {
    businessId
    name
    address
    avgStars
  }
}
```

```
{
  "data": {
    "businessBySearchTerm": [
      {
        "businessId": "b1",
        "name": "Missoula Public Library",
        "address": "301 E Main St, Missoula, MT 59802",
        "avgStars": 3.5
      },
      {
        "businessId": "b2",
        "name": "San Mateo Public Library",
        "address": "55 W 3rd Ave, San Mateo, CA 94402",
        "avgStars": 5
      }
    ]
  }
}
```

Рис. 2.5. Запрос с условием

А потом получим отзывы для каждой найденной компании (рис. 2.6).

```
{
  businessBySearchTerm(search: "Library") {
    businessId
    name
    address
    avgStars
    reviews {
      stars
      text
    }
  }
}
```

```
{
  "data": {
    "businessBySearchTerm": [
      {
        "businessId": "b1",
        "name": "Missoula Public Library",
        "address": "301 E Main St, Missoula, MT 59802",
        "avgStars": 3.5,
        "reviews": [
          {
            "stars": 3,
            "text": "Friendly staff. Interlibrary loan is super fast"
          },
          {
            "stars": 4,
            "text": "Easy downtown access, lots of free parking"
          }
        ]
      },
      {
        "businessId": "b2",
        "name": "San Mateo Public Library",
        "address": "55 W 3rd Ave, San Mateo, CA 94402",
        "avgStars": 5,
        "reviews": [
          {
```

Рис. 2.6. Добавление поля reviews в запрос

Полный код примера можно найти в репозитории книги на GitHub: **http://mng.bz/J2jo**. В следующей главе мы познакомимся с графовой базой данных Neo4j и узнаем, как моделировать, хранить и запрашивать данные с помощью языка запросов Cypher.

2.4. Упражнения

1. Изучите другие требования к нашему приложению, которые мы еще не реализовали. Сможете ли вы написать запросы GraphQL для удовлетворения этих требований? Что получилось в результате?
2. В каких других полях в нашем API следует использовать разбиение на страницы? Обновите определения типов, включив соответствующие поля для упорядочения и разбиения на страницы, и обновите функции разрешения для обработки соответствующих аргументов.
3. Реализуйте корневую функцию разрешения для `usersById`.
4. В нашем примере GraphQL API явно не хватает категорий компаний. Обновите примеры данных, определения типов GraphQL и функции разрешения для добавления категорий. Подумайте, как можно смоделировать категории в API, учитывая, что поиск по категориям определен как отдельное требование.

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

Итоги

- Моделируя API, следует учитывать требования к приложению. В таком случае, как результат отображения ментальной модели данных, создается граф, узлы которого – это сущности, а ребра – отношения, соединяющие их.
- Типы GraphQL определяют данные, отношения и точки входа в GraphQL API. Типы определяются на языке определения схем (SDL), не зависящем ни от какого другого языка программирования. GraphQL поддерживает множество встроенных типов (`ID`, `String`, `Int`, `Float`, `Bool` и т. д.), а также позволяет определять нестандартные скаляры и типы.
- Функции разрешения (resolvers) содержат логику выборки данных из GraphQL API. Функции разрешения вызываются каскадным образом, в зависимости от набора запрашиваемых полей. Каждой функции разрешения передается объект контекста, который может содержать строку подключения к базе данных или другие вспомогательные объекты для доступа к данным.
- Apollo Server используется для объединения определений и функций разрешения в выполняемую схему GraphQL и обслуживания GraphQL API.
- Apollo Studio можно использовать для просмотра схемы GraphQL API, а также для выполнения запросов и просмотра результатов.

Глава 3

Графы в базе данных

В этой главе:

- введение в графовые базы данных, в частности в Neo4j;
- свойства графовой модели данных;
- использование языка запросов Cypher для создания и получения данных из Neo4j;
- использование клиентских драйверов для Neo4j, в частности драйвера для JavaScript Node.js.

По сути, графовая база данных – это программный инструмент, позволяющий пользователю моделировать, хранить и запрашивать данные в виде графа. Организация хранилища в виде графа часто упрощает моделирование сложных связанных данных и обеспечивает более высокую эффективность работы со сложными запросами, требующими обхода множества связанных сущностей.

В этой главе мы приступим к созданию графовой модели данных, опираясь на требования из предыдущей главы, и сравним ее со схемой GraphQL, созданной там же. Мы также начнем изучать язык запросов Cypher, уделив особое внимание приемам написания запросов Cypher для удовлетворения требований к нашему приложению. Попутно будет показано, как установить Neo4j, как использовать Neo4j Desktop для создания новых проектов Neo4j и как использовать Neo4j Browser для отправки запросов в Neo4j и визуализации ответов. Наконец, мы увидим, как использовать клиентский драйвер Neo4j JavaScript в простом приложении Node.js, которое запрашивает данные из Neo4j.

3.1. Обзор Neo4j

Neo4j – это графовая база данных, использующая графовую модель свойств для представления данных и поддерживающая язык запросов Cypher для взаимодействия с ней. Neo4j – это транзакционная база данных с полными гарантиями ACID (Atomicity, Consistency, Isolation, Durability – атомарность, согласованность, изоляция, надежность), необходимыми большинству рабочих нагрузок, а также может использоваться для анализа графов. Графовые базы данных, такие как

Neo4j, оптимизированы для работы с тесно связанными данными и запросами, для выполнения которых требуется производить обход графа (эту ситуацию можно сравнить с несколькими операциями `JOIN` в реляционной базе данных). По этой причине графовые базы данных можно считать идеальными хранилищами для GraphQL API, который описывает связанные данные и часто генерирует сложные вложенные запросы. Neo4j распространяется с открытым исходным кодом и доступна для загрузки по адресу **https://neo4j.com/download/**.

В этой главе мы будем учиться создавать и запрашивать данные в Neo4j с помощью Neo4j Desktop и Neo4j Browser, но прежде я предлагаю поближе рассмотреть графовую модель свойств, используемую в Neo4j, и понять, как она соотносится с моделью, используемой для описания GraphQL API.

3.2. Моделирование графовых данных в Neo4j

В отличие от других баз данных, представляющих данные в виде таблиц или документов, графовые базы данных, такие как Neo4j, моделируют, хранят и позволяют запрашивать данные в виде графа. Узлы в графе – это объекты, а ребра, соединяющие их, – это отношения. В реляционной базе данных отношения моделируются с помощью внешних ключей и операций соединения. В документной базе данных для ссылок на другие сущности используются идентификаторы, а иногда даже приходится идти по пути денормализации и встраивать другие сущности в один документ (рис. 3.1).

Рис. 3.1. Сравнение реляционной, документной и графовой моделей данных

Первый шаг при работе с базой данных – определение модели данных. В нашем случае модель данных будет основываться на требованиях, которые мы определили в предыдущей главе, и использоваться для работы с компаниями, пользователями и отзывами. Просмотрите требования, перечисленные в первом разделе предыдущей главы, чтобы освежить память. Возьмем эти требования и, вооружившись знанием предметной области, создадим модель на доске.

Модель на доске

Под термином «модель на доске» мы будем подразумевать диаграмму, какие обычно создаются при первых рассуждениях о предметной области. Такие модели часто представляют собой граф сущностей и взаимосвязей между ними, нарисованный на доске, как показано на рисунке ниже.

Графовая модель свойств: модель на доске

Как преобразовать эту ментальную модель на доске в физическую модель данных, используемую базой данных? В других системах для этого иногда создают диаграммы сущность–связь (Entity-Relationship, ER), т. е. определяют схему базы данных. Про Neo4j говорят, что она *не требует наличия схемы данных*. Мы можем определить ограничения базы данных, такие как уникальность свойств, чтобы обеспечить их соблюдение, но точно так же можно использовать Neo4j без этих ограничений или схемы. И все же первым шагом нужно определить модель в виде графовой модели свойств, которая используется в Neo4j и в других графовых базах данных. Давайте преобразуем нашу простую модель на доске, изображенную на рисунке во врезке выше, в графовую модель свойств, которую можно использовать в базе данных.

3.2.1. *Графовая модель свойств*

В главе 1 уже был дан краткий обзор графовой модели свойств, а теперь мы рассмотрим процесс преобразования нашей модели на доске в графовую модель свойств, используемую базой данных.

Графовая модель свойств

Графовая модель свойств состоит из:

- меток узлов – меток сущностей или объектов в модели данных. Узел может иметь одну или несколько меток, описывающих порядок их группировки (метки можно рассматривать как типы);
- отношения (связи) – каждое отношение связывает два узла и имеет один тип и направление;
- свойства – произвольные пары ключ/значение, хранящиеся в узлах или отношениях.

Метки узлов

Узлы – это объекты в нашей модели на доске. Каждый узел может иметь одну или несколько меток, описывающих порядок группировки узлов. Обычно метки

добавляются просто, потому что группировка уже определена в процессе рисования модели на доске – мы лишь формализуем описания, используемые для ссылки на узлы, в метки узлов. Позже мы добавим псевдонимы узлов и дополнительные метки, поэтому для обозначения метки мы используем двоеточие в качестве разделителя (рис. 3.2).

Рис. 3.2. Графовая модель свойств: метки узлов

Инструменты создания диаграмм, изображающих графовые модели данных

Существует множество инструментов для построения диаграмм графовых моделей данных. В этой книге мы используем инструмент **Arrows** – простое веб-приложение, позволяющее создавать графовые модели данных. Это приложение доступно онлайн по адресу **https://arrows.app.**

Приложение Arrows имеет минималистичный пользовательский интерфейс и предназначено для создания графовых моделей свойств:

- создайте новые узлы с помощью кнопки **Add Node** (Добавить узел) или перетаскивая мышью существующие;
- для создания связи наведите указатель мыши на границу узла и, ухватив за появившийся ореол, перетащите связь в пустое место, чтобы создать новый связанный узел, или в центр существующего узла;
- выполните двойной щелчок мышью на узле или на связи, чтобы отредактировать их, определить имена и свойства (в форме **ключ: значение**);
- по окончании создания диаграммы ее можно экспортировать в PNG, SVG и другие форматы (включая определения типов GraphQL).

Для меток узлов используется соглашение `PascalCase`. Дополнительные примеры соглашений об именах вы найдете в руководстве по стилю Cypher на сайте **https://neo4j.com/developer/cypher/style-guide/**. Узлы могут иметь несколько меток и позволяют определять иерархии типов, роли в разных контекстах или даже мультиарендность.

Отношения

Следующий шаг после определения меток узлов – определение отношений (связей) в модели данных. Отношения имеют один тип и направление, но могут запрашиваться в любом направлении (как показано на рисунке во врезке ниже).

Работа с ненаправленными отношениями

Каждая связь имеет одно направление, но вообще их можно считать ненаправленными и не указывать направление в запросах Cypher.

Графовая модель свойств: типы отношений

Имена для отношений должны отражать тип перехода от одного узла к другому узлу и вместе с именами узлов должны читаться как простые предложения, например `User wrote review` (пользователь написал отзыв) или `Review reviews business` (просмотр отзывов о компании). Дополнительные примеры соглашений об именах вы найдете в руководстве по стилю Cypher: **https://neo4j.com/developer/cypher/style-guide/**.

Свойства

Свойства – это произвольные пары ключ/значение, хранящиеся в узлах и связях и представляющие атрибуты или поля сущностей в модели данных. В нашем примере мы создадим в узле `User` строковые свойства `userId` и `name`, а также другие свойства в узлах `Review` и `Business`.

Работа с ненаправленными отношениями

Каждая связь имеет одно направление, но вообще их можно считать ненаправленными и не указывать направление в запросах Cypher.

- `String`;
- `Float`;
- `Long`;
- `Date`, `DateTime` и другие типы для работы с датами и временем;
- списки предыдущих типов.

Графовая модель свойств: свойства

3.2.2. *Ограничения базы данных и индексы*

После определения модели данных мы можем использовать ее в базе данных. Но как это сделать? Как упоминалось выше, в отличие от других баз данных, требующих определения полной схемы перед добавлением данных, Neo4j не требует наличия заранее определенной схемы данных. Вместо этого можно определить ограничения, гарантирующие соответствие данных правилам предметной области. Например, можно задать ограничение уникальности, гарантирующее уникальность значений свойств метки узла (например, что никакие два пользователя не будут иметь повторяющееся значение свойства `ID`), ограничение существования свойств (например, что набор свойств существует на момент создания или изменения узла или связи) и ограничения ключа узла, аналогичного составному ключу и создающего ограничение на основе нескольких свойств.

Ограничения базы данных поддерживаются индексами, которые можно создавать отдельно. В графовой базе данных индексы используются для поиска начальной точки обхода, а не для обхода графа. Более подробно ограничения базы данных и индексы мы рассмотрим в следующем разделе, посвященном Cypher.

3.3. Вопросы моделирования данных

Моделирование графовых данных – итеративный процесс. Вот как он выглядит в общих чертах.

1. Какие сущности имеются? Как они группируются? По ответам на эти вопросы создаются узлы и метки узлов.
2. Как связаны эти сущности? По ответам на этот вопрос создаются отношения.
3. Какие атрибуты имеют узлы и отношения? По ответам на этот вопрос создаются свойства.
4. Можно ли определить порядок обхода графа, отвечающий на ваши вопросы? По ответам на этот вопрос создаются запросы Cypher. Если ответа на вопрос нет, повторите итерацию.

Однако часто встречаются нюансы, не предусмотренные этим общим подходом. В следующем разделе мы рассмотрим некоторые из них.

3.3.1. *Узел и свойство*

Иногда бывает сложно определить, как должно моделироваться значение – как узел или как свойство узла. В таких случаях попробуйте задать себе вопрос: «Могу ли я обнаружить что-то полезное, пройдя через это значение, если бы это был узел?» Если да, то это значение следует смоделировать как узел; если нет, то лучше превратить его в свойство. Например, рассмотрим задачу добавления категории компаний в нашу модель. Поиск компаний в пересекающихся категориях может очень пригодиться, и их проще обнаруживать, если смоделировать их как узлы. С другой стороны, юридический адрес компании. Если смоделировать адрес как узел, а не как свойство, то какую пользу может дать обход узлов с адресами? В подавляющем большинстве случаев в этом нет смысла, поэтому лучше смоделировать это значение как свойство.

3.3.2. *Узел и отношение*

В случае, когда есть данные, связывающие два узла (например, отзыв о компании, написанный пользователем), возникает вопрос, как их моделировать – как узлы или как отношения? На первый взгляд кажется, что можно просто создать связь `REVIEWS`, соединяющую пользователя с компанией, и сохранить информацию об отзыве, такую как количество звезд и текст, в виде свойств отношения. Однако впоследствии может появиться потребность извлечь данные из обзора, например ключевые слова, с помощью некоторого приема обработки естественного языка и связать эти извлеченные данные с отзывом. Или может понадобиться использовать узлы отзывов в качестве отправной точки для обхода. Эти два примера наглядно показывают, почему эти данные желательно смоделировать как промежуточный узел, а не как отношение.

3.3.3. *Индексы*

Индексы используются в графовых базах данных для поиска начальной точки обхода, но не во время обхода. Это важная особенность работы графовых баз данных, таких как Neo4j, известная как *безындексная смежность*. Индексы должны создаваться только для свойств, которые будут использоваться для поиска начальной точки обхода, таких как имя пользователя или идентификатор компании.

3.3.4. *Специфика типов отношений*

Типы отношений (связей) определяют способ группировки отношений и должны передавать ровно столько информации, чтобы было ясно, как связаны два узла, но без лишней конкретики. Например, `REVIEWS` – это хороший тип отношений, соединяющий узлы `Review` и `Business`. `REVIEW_WRITTEN_BY_BOB_FOR_PIZZA` – это слишком конкретный тип отношений; имя пользователя и ресторан хранятся в другом месте и не должны дублироваться в типе отношения.

3.3.5. *Выбор направления отношений*

Все отношения (связи) в графовой модели свойств имеют одно направление, но могут опрашиваться в любом направлении. Нет необходимости создавать повторяющиеся отношения для моделирования двунаправленности. В общем случае старайтесь выбирать такие направления отношений, которые обеспечивают последовательное чтение модели данных.

3.4. Инструменты: Neo4j Desktop

Теперь, выяснив, как конструируется графовая модель свойств, и определив простую версию модели, которая будет использоваться в нашем приложении, создадим базу данных Neo4j и попробуем выполнить некоторые запросы Cypher. Для этого воспользуемся приложением Neo4j Desktop, которое является центром управления Neo4j (рис. 3.3). В Neo4j Desktop можно создавать проекты и экземпляры Neo4j, запускать, останавливать и настраивать экземпляры базы данных Neo4j, а также устанавливать дополнительные плагины, такие как Graph Data Science и APOC (стандартная библиотека процедур для базы данных Neo4j). Neo4j Desktop

также включает функции для установки *графовых приложений*, которые запускаются в Neo4j Desktop и подключаются к активному экземпляру Neo4j. Одним из примеров таких приложений может служить браузер Neo4j Browser, устанавливаемый по умолчанию. Примеры других графовых приложений можно найти на сайте **https://install.graphapp.io/**.

Рис. 3.3. Neo4j Desktop: центр управления Neo4j

Если вы еще не установили Neo4j Desktop, сделайте это прямо сейчас, загрузив приложение с сайта **https://neo4j.com/download/**. Neo4j Desktop доступен для систем Mac, Windows и Linux.

После загрузки и установки Neo4j Desktop создайте новый локальный экземпляр Neo4j, выбрав пункт **Add Graph** (Добавить граф). Вам будет предложено ввести имя базы данных и пароль. Пароль может быть каким угодно; просто не забудьте запомнить его. Создав граф, щелкните на кнопке **Start** (Пуск), чтобы активировать его. Далее мы будем использовать браузер Neo4j Browser для выполнения запросов к только что созданной базе данных.

3.5. Инструменты: Neo4j Browser

Neo4j Browser – это среда для работы с Neo4j, которая позволяет взаимодействовать с базой данных, создавать запросы Cypher и просматривать результаты (рис. 3.4). Запустите Neo4j Browser, щелкнув на значке приложения в разделе **Application** (Приложения) в Neo4j Desktop.

Рис. 3.4. Neo4j Browser: среда для работы с запросами Cypher и Neo4j

Neo4j Browser позволяет запускать запросы Cypher к Neo4j и обрабатывать результаты. Но, прежде чем углубиться в приемы использования Neo4j Browser, рассмотрим язык запросов Cypher.

3.6. Cypher

Cypher – это декларативный язык запросов, поддерживаемый графовыми базами данных и обладающий некоторыми возможностями, напоминающими SQL. Фактически Cypher можно считать *языком SQL для графов*. Cypher поддерживает сопоставление с образцом с использованием специальной нотации для описания графовых шаблонов. В этом разделе мы рассмотрим некоторые основные возможности Cypher, связанные с созданием и получением данных, включая использование предикатов и агрегатов. Мы охватим лишь небольшую часть языка Cypher; за более полной информацией обращайтесь к справочной таблице **https://neo4j.com/docs/cypher-refcard/current/** или к документации, доступной по адресу **https://neo4j.com/docs/cypher-manual/current/**.

3.6.1. *Сопоставление с образцом*

Сопоставление с образцом является основным средством для любого декларативного языка запросов. В Cypher это средство используется и для создания, и для получения данных. Вместо передачи базе данных точных операций (императивный стиль) мы описываем на языке Cypher шаблон (или образец) того, что нужно найти или создать, а база данных выполняет соответствующие последовательности операций, которые удовлетворяют инструкции. В основе этой декларативной функциональности лежит описание графовых шаблонов с использованием специальной нотации (также называемой мотивами).

Узлы

Узлы определяются в круглых скобках `()`. При желании можно указать метку(и) узла, используя двоеточие в качестве разделителя, например `(:User)`.

Отношения

Отношения определяются в квадратных скобках `[]`. При желании можно указать тип и направление:

```
(:Review)-[:REVIEWS]->(:Business).
```

3.6.2. *Свойства*

Свойства задаются парами «имя: значение», перечисленными через запятую в фигурных скобках `{}`. Примером свойств может служить название компании или имя пользователя.

Псевдонимы

К элементам графа могут быть привязаны псевдонимы (переменные), на которые можно ссылаться в запросе. Например, в шаблоне `(r:Review)-[a:REVIEWS]->(b:Business)` псевдоним `r` становится переменной, привязанной к узлу `Review`,

псевдоним a привязывается к отношению REVIEWS, а псевдоним b – к узлу Business. Эти переменные доступны только в запросе Cypher, где они определены. Следуйте инструкциям, выполняя запросы Cypher в Neo4j Browser, когда мы будем представлять команды Cypher для создания и получения данных, соответствующих модели данных, построенной в этой главе.

3.6.3. CREATE

Первое, что нужно сделать, – создать некоторые данные в базе данных с помощью команды CREATE. Для начала создадим в графе один узел Business, выполнив команду CREATE, за которой следует шаблон, описывающий добавляемые данные:

Ниже показан результат выполнения этой команды в Neo4j Browser:

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

Это сообщение означает, что создан один узел с новой меткой и установлено одно свойство узла – в данном случае свойство name узла с меткой Business. Также для настройки свойства можно использовать команду SET. Следующий пример эквивалентен предыдущему:

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

Для отображения созданных данных можно добавить команду RETURN, которая отобразит полученный результат в Neo4j Browser в виде графа. Например, следующий код

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

отобразит в Neo4j Browser результат, как показано на рис. 3.5.

В команде CREATE можно использовать более сложные шаблоны, например определяющие отношения. Обратите внимание на нотацию определения отношения с помощью квадратных скобок <-[]-, включающую направление отношения (рис. 3.6):

```
CREATE (b:Business)<-[:REVIEWS]-(r:Review)
SET b.name = "Bob's Pizza",
```

```
    r.stars = 4,
    r.text = "Great pizza"
RETURN b, r
```

Рис. 3.5. Создание узла в Neo4j Browser с помощью Cypher

Рис. 3.6. Результат создания двух узлов и отношения

Рис. 3.7. Создание подграфа

Язык Cypher позволяет передавать графовые шаблоны произвольной сложности. В следующем примере дополнительно определяется пользователь, связанный с отзывом (рис. 3.7):

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

Обратите внимание, что этот запрос связывает весь графовый шаблон с переменной `p` и возвращает эту переменную. В данном случае `p` получит значение, отражающее весь созданный путь (комбинацию узлов и связей).

До сих пор мы возвращали только данные, созданные командой Cypher. Но как запросить и отобразить остальные данные, имеющиеся в базе данных? Для этого следует использовать ключевое слово `MATCH`. Давайте отыщем все узлы в базе данных и вернем их:

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

В Neo4j Browser должен появиться граф, изображенный на рис. 3.8.

Как видите, здесь явно что-то пошло не так; мы создали много повторяющихся узлов! Удалим все данные из базы:

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

Рис. 3.8. Созданы повторяющиеся узлы

Этот образец будет соответствовать всем узлам, и команда удалит узлы, а также все отношения между ними. В Neo4j Browser должно появиться сообщение, перечисляющее удаленные сущности:

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

Теперь начнем сначала и посмотрим, как создавать данные без дубликатов.

3.6.4. *MERGE*

Чтобы избежать появления дубликатов, можно использовать команду `MERGE`. Она создает данные, указанные в шаблоне, только если они еще отсутствуют в базе данных. Перед использованием `MERGE` хорошо бы создать ограничение уникальности для свойства, имеющего уникальные значения, например для поля ID. При создании ограничения уникальности также будет создан индекс. Пример создания ограничения уникальности будет показан в следующем разделе. В простых случаях можно использовать `MERGE` без этих ограничений, поэтому вернемся к команде Cypher, создавшей узлы `Business`, `Review` и `User`, но на этот раз используем `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 *
```

На рис. 3.9 показан получившийся граф с созданными данными.

Рис. 3.9. Результат использования MERGE для создания данных

Результаты этой команды Cypher выглядят так же, как и результаты предыдущей версии с использованием `CREATE`; однако есть одно важное отличие: этот запрос – *идемпотентный*. Сколько бы раз он ни запускался, он не создаст повторяющихся узлов, потому что в нем используется `MERGE` вместо `CREATE`. Мы еще вернемся к `MERGE` в следующей главе, где поговорим о том, как создавать данные с помощью GraphQL API.

Индексы в Neo4j

Важно понимать, как используются индексы в графовых базах данных, таких как Neo4j. Выше отмечалось, что Neo4j имеет свойство, называемое безындексной смежностью, согласно которому переход от узла к любому другому узлу не требует поиска по индексу. Так как же индексы используются в Neo4j? Индексы используются только для поиска начальной точки обхода. Это отличает графовые базы данных от реляционных, где индексы применяются для вычисления перекрытия наборов (таблиц). Графовые базы данных просто вычисляют смещения в хранилище, по сути, следуя по указателям, которые, как мы знаем, компьютеры обрабатывают очень быстро.

3.6.5. *Определение ограничений на Cypher*

Выше упоминалось об ограничениях базы данных и их связи с (необязательным) определением схемы в Neo4j, когда создавали модель данных. Далее мы рассмотрим синтаксис Cypher для создания ограничений, соответствующих нашей модели данных.

Ограничение уникальности

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

Ограничение существования свойства

```
CREATE CONSTRAINT ON (b:Business) ASSERT b.businessId IS NOT NULL
```

Ограничение ключа узла

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

Обратите внимание, что если в базе данных имеются повторяющиеся данные, конфликтующие с любым из этих ограничений, вы получите сообщение об ошибке, предупреждающее о невозможности создать ограничение. В таком случае можно удалить все данные, а затем снова попытаться создать ограничение.

3.6.6. *MATCH*

Теперь после добавления данных в граф можно начинать писать запросы для удовлетворения некоторых требований к нашему приложению. Оператор `MATCH` похож на `CREATE`. Он также принимает шаблон, но дополнительно может содержать оператор `WHERE`, определяющий предикаты для фильтрации результатов применения шаблона. Оператор `MATCH` используется для поиска данных в базе данных, соответствующих заданному шаблону. Вот пример поиска всех пользователей в базе данных:

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

Конечно, в `MATCH` можно использовать более сложные шаблоны:

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

Этот запрос вернет всех пользователей, написавших хотя бы один отзыв о любой компании. А что, если понадобится отыскать только отзывы об определенной компании? В этом случае нужно добавить соответствующие предикаты, используя оператор `WHERE`.

WHERE

Оператор `WHERE` позволяет добавлять предикаты в оператор `MATCH`. Чтобы найти компанию «Bob's Pizza», например, можно написать следующий запрос Cypher:

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

Для сравнения на равенство можно использовать более краткую форму записи:

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

3.6.7. Агрегаты

Часто бывает нужно вычислить агрегированные значения по набору результатов, например средний рейтинг всех отзывов о компании «Bob's Pizza». Для этого можно использовать агрегатную функцию `avg`:

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

После запуска этого запроса в Neo4j Browser вместо графа появится таблица с результатами, потому возвращаются не графовые, а табличные данные:

"avg(r.stars)"
4.0

А можно ли получить средний рейтинг для каждой компании? В SQL-запросе для этой цели можно было бы использовать оператор `GROUP BY`, чтобы с его помощью сгруппировать отзывы по названиям компаний и вычислить агрегатное значение по каждой группе, но в Cypher нет оператора `GROUP BY`. Вместо этого в Cypher используется *неявная группировка* по операциям. Например, вот как выглядит запрос на языке Cypher, вычисляющий средний рейтинг для каждой компании:

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

В нашем примере он вернет следующую таблицу:

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

Результаты выглядят не особенно интересными, потому что в нашей базе данных только одна компания и один отзыв. В разделе с упражнениями для этой главы вам будет предложено поработать с более объемным набором данных.

3.7. Использование клиентских драйверов Neo4j

До сих пор для выполнения запросов Cypher мы использовали Neo4j Browser, что довольно удобно для оценки возможностей на начальном этапе и прототипирования; однако чаще нашей целью является создание приложения, программно взаимодействующего с базой данных. В таких приложениях используются клиентские драйверы Neo4j. Эти драйверы доступны для многих языков, таких как JavaScript, Java, Python, .NET и Go, и позволяют разработчику выполнять запросы Cypher к экземпляру Neo4j с помощью согласованного API, идиоматичного для конкретного языка программирования. В главе 1 был показан пример использования драйвера Neo4j JavaScript для выполнения запроса Cypher и обработки результатов. Дополнительные сведения о драйверах Neo4j вы найдете в руководствах по драйверам и языкам: **https://neo4j.com/developer/language-guides/**.

В следующей главе мы вооружимся идеями и инструментами, обсуждавшимися до сих пор (GraphQL и Neo4j), и создадим GraphQL API, который использует Neo4j в качестве хранилища. Для этого мы задействуем библиотеку Neo4j GraphQL, упрощающую и ускоряющую процесс создания GraphQL API, поддерживаемых Neo4j.

3.8. Упражнения

Перед выполнением следующих упражнений запустите следующую команду в Neo4j Browser, чтобы загрузить руководство со встроенными запросами Cypher: `:play grandstack`. Это руководство проведет вас через процесс загрузки более крупного и полного набора данных с компаниями и отзывами. После загрузки данных в Neo4j переходите к упражнениям.

1. Запустите команду `CALL db.schema.visualization()`, чтобы проверить модель данных. Какие метки узлов и какие типы отношений используются?
2. Напишите запрос Cypher, возвращающий список всех пользователей в базе данных. Сколько всего пользователей? Как их зовут?
3. Найдите все отзывы, написанные пользователем по имени `Will`. Вычислите среднюю оценку этого пользователя.
4. Найдите все компании, для которых пользователь `Will` оставил отзыв. В какой категории компаний больше всего отзывов этого пользователя?
5. Напишите запрос, возвращающий пользователю `Will` рекомендации с названиями компаний, для которых он еще не написал отзывы.

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

Итоги

- Графовая база данных позволяет моделировать, хранить и запрашивать данные в виде графов.
- Графовая модель свойств используется графовыми базами данных и состоит из меток узлов, связей и свойств.
- Язык запросов Cypher – это декларативный язык запросов, ориентированный на сопоставление с образцом и используемый для описания запросов к графовым базам данных, включая Neo4j.
- Клиентские драйверы используются для создания приложений, взаимодействующих с Neo4j. Драйверы позволяют приложениям отправлять запросы Cypher в базу данных и обрабатывать результаты.

Глава 4

Библиотека Neo4j GraphQL

В этой главе:

- обзор проблем, часто возникающих при создании приложений GraphQL API;
- средства интеграции с базой данных для поддержки GraphQL, помогающие решать эти проблемы, включая библиотеку Neo4j GraphQL;
- создание конечной точки GraphQL на основе Neo4j с использованием преимуществ библиотеки Neo4j GraphQL, таких как генерирование запросов типа `query` и `mutation`, фильтрация, а также использование временных и пространственных типов данных;
- расширение возможностей автоматически сгенерированного GraphQL API с помощью пользовательской логики;
- анализ схемы GraphQL в существующей базе данных Neo4j.

Реализации GraphQL сталкиваются с некоторыми типичными проблемами, негативно влияющими на производительность приложений и продуктивность разработчиков. Мы уже видели некоторые из этих проблем выше (например, проблема *n* + 1 запросов), и в этой главе мы более подробно рассмотрим эти проблемы и обсудим приемы их смягчения с использованием средств интеграции с базой данных для поддержки GraphQL, которые упрощают создание эффективных GraphQL API.

В частности, мы рассмотрим библиотеку Neo4j GraphQL для Node.js, предназначенную для работы с реализациями GraphQL на JavaScript, такими как Apollo Server, и для создания GraphQL API, поддерживаемых Neo4j. Библиотека Neo4j GraphQL позволяет создавать полноценные GraphQL API из определений типов GraphQL, управлять моделью данных базы данных и автоматически генерировать функции разрешения для выборки и изменения данных, включая сложную фильтрацию, упорядочение и разбиение на страницы. Библиотека Neo4j GraphQL также позволяет добавлять пользовательскую логику к автоматически сгенерированным операциям создания, чтения, обновления и удаления.

Знакомясь с библиотекой Neo4j GraphQL на примере нашего GraphQL API для приложения обзора компаний, мы добавим слой доступа к хранилищу Neo4j. При

этом основное внимание мы будем уделять получению существующих данных, используя учебный набор данных в Neo4j из предыдущей главы. А создание и обновление данных (мутации в терминологии GraphQL), а также более сложную семантику запросов GraphQL, такую как интерфейсы и фрагменты, мы изучим в последующих главах, где эти концепции будут представлены в контексте построения пользовательского интерфейса. На рис. 4.1 показано, как библиотека Neo4j GraphQL вписывается в общую архитектуру нашего приложения. Цель библиотеки Neo4j GraphQL – упростить создание API, поддерживаемого Neo4j.

Рис. 4.1. Библиотека Neo4j GraphQL помогает создать слой API между клиентом и базой данных

4.1. Распространенные проблемы GraphQL

При создании GraphQL API разработчики обычно сталкиваются с двумя типами проблем: низкая производительность и необходимость писать большой объем типового кода, что может повлиять на продуктивность разработчиков.

4.1.1. *Низкая производительность и проблема n + 1 запросов*

Выше мы уже обсуждали проблему $n + 1$ запросов, возникающую, когда хранилищу данных посылается несколько запросов для разрешения одного запроса GraphQL. Из-за каскадного способа вызова функций разрешения GraphQL часто требуется выполнить несколько запросов к базе данных. Например, представьте запрос, выбирающий компании по названию, а также все отзывы для каждой из них. Простейшая реализация сначала запросила бы в базе данных все компании, соответствующие критериям поиска. Затем для каждой компании отправила бы дополнительный запрос, чтобы найти все отзывы о ней. Каждый запрос к базе данных вызывает дополнительную задержку, необходимую для его передачи по сети, что может значительно повлиять на производительность.

Распространенное решение этой проблемы – использование шаблона кеширования и пакетной обработки, известного как DataLoader (Загрузчик данных). Этот шаблон способен решить некоторые проблемы с производительностью, но и он может потребовать выполнить несколько запросов к базе данных и не может использоваться во всех случаях, например когда идентификатор объекта неизвестен.

4.1.2. *Типовой код и продуктивность разработчиков*

Под *типовым кодом* подразумевается повторяющийся код, решающий общие задачи. В процессе реализации GraphQL API часто требуется писать типовой код

для реализации логики выборки данных в функциях разрешения. Это может отрицательно сказаться на продуктивности разработчиков и замедлить разработку, потому что разработчику приходится писать простую логику выборки данных для каждого типа и поля, вместо того чтобы сосредоточиться на ключевых компонентах приложения. В контексте нашего приложения это означало бы написание логики поиска компаний по названию в базе данных, поиска отзывов, связанных с каждой компанией, и каждого пользователя, написавшего каждый отзыв, и т. д., пока не будет определена вся логика получения всех полей, объявленных в нашей схеме GraphQL.

4.2. Введение в средства интеграции GraphQL с базой данных

Средства интеграции GraphQL с базами данных – это класс инструментов, позволяющих создавать GraphQL API, взаимодействующие с базами данных. Существует несколько таких инструментов, отличающихся наборами возможностей и уровнями интеграции, – в этой книге мы сосредоточимся на библиотеке Neo4j GraphQL. Однако в общем случае цель этих средств состоит в последовательном решении общих проблем GraphQL, выявленных ранее, за счет сокращения объема типового кода и времени, необходимого на выборку данных.

В оставшейся части этой главы основное внимание будет уделено библиотеке Neo4j GraphQL, предназначенной для создания GraphQL API с поддержкой Neo4j. Важно отметить, что наш GraphQL API служит прослойкой между клиентом и базой данных – мы не будем напрямую запрашивать базу данных со стороны клиента. Слой API играет важную роль, позволяя реализовывать функции, такие как авторизация, и пользовательскую логику. Кроме того, поскольку GraphQL – это язык запросов API (а не язык запросов к базе данных), ему не хватает некоторых черт (например, проекций), которые мы ожидаем от языка запросов к базе данных.

4.3. Библиотека Neo4j GraphQL

Библиотека Neo4j GraphQL – это библиотека Node.js, работающая с любой реализацией GraphQL на JavaScript, такой как GraphQL.js и Apollo Server, и предназначенная для максимального упрощения создания GraphQL API с поддержкой базы данных Neo4j. Двумя основными особенностями библиотеки Neo4j GraphQL являются создание схемы GraphQL и преобразование запросов GraphQL в запросы Cypher. Документация проекта доступна по адресу **http://mng.bz/woNO**.

Преобразование запросов GraphQL в запросы Cypher позволяет:

- сгенерировать один запрос к базе данных на основе произвольного запроса GraphQL;
- выполнять пользовательскую логику, определенную в схеме GraphQL, в виде подзапросов, включенных в сгенерированные запросы к базе данных.

Процесс генерирования схемы GraphQL принимает определения типов GraphQL и генерирует GraphQL API с операциями создания, чтения, обновления, удале-

ния (Create, Read, Update, Delete – CRUD) для определенных типов. В семантике GraphQL это означает добавление в схему типов `Query` и `Mutation` и создание функций разрешения для них. Сгенерированный API включает поддержку фильтрации, упорядочивания, разбиения на страницы и собственных типов баз данных, таких как пространственные и временные типы, без необходимости определять их вручную. Результатом процесса является выполняемый объект схемы GraphQL, который затем можно передать реализации сервера GraphQL, такой как Apollo Server, для обслуживания GraphQL API. Процесс генерирования схемы избавляет от необходимости писать типовой код для выборки данных и отображения схемы GraphQL в схему базы данных.

Процесс преобразования GraphQL происходит во время выполнения запроса. При получении запроса GraphQL генерируется один запрос Cypher, отправляемый в базу данных. Генерирование одного запроса к базе данных для любой произвольной операции GraphQL решает проблему $n + 1$ запросов, гарантируя только одно обращение к базе данных для любого запроса GraphQL. Документацию с описанием библиотеки Neo4j GraphQL и ссылки на другие ресурсы можно найти по адресу **https://neo4j.com/product/graphql-library/**.

4.3.1. *Настройка проекта*

Далее в этой главе мы исследуем возможности библиотеки Neo4j GraphQL, создав новый GraphQL API для Neo4j и используя учебный набор данных, который можно загрузить, как описывалось в разделе «Упражнения» в предыдущей главе. Сначала создадим новый проект Node.js, использующий библиотеку Neo4j GraphQL и драйвер Neo4j JavaScript для извлечения данных из Neo4j. Затем исследуем различные возможности библиотеки Neo4j GraphQL, добавляя свой код в приложение GraphQL API по мере продвижения.

Neo4j

Прежде всего запустите экземпляр Neo4j (для этого можно использовать Neo4j Desktop, Neo4j Sandbox или Neo4j Aura, но мы будем считать, что вы используете Neo4j Desktop). Пользователям Neo4j Desktop необходимо установить плагин из стандартной библиотеки APOC. Пользователи Neo4j Sandbox или Neo4j Aura могут пропустить этот шаг, потому что APOC включен в эти службы по умолчанию. Чтобы установить APOC в Neo4j Desktop, перейдите на вкладку **Plugins** (Плагины) в проекте, затем найдите APOC в списке доступных плагинов, выберите его и щелкните на кнопке **Install** (Установить). Потом убедитесь, что база данных Neo4j пуста, выполнив оператор Cypher в листинге 4.1.

> **ВНИМАНИЕ.** Этот оператор удалит все данные в базе данных Neo4j, поэтому убедитесь, что это именно тот экземпляр, который вы предполагаете использовать для следования за примерами, а не какой-то другой.

Листинг 4.1. Очистка базы данных Neo4j

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

Теперь мы готовы загрузить учебный набор данных, что вы, возможно, уже сделали, если выполнили упражнения в предыдущей главе. Запустите следующую команду в Neo4j Browser (рис. 4.2):

```
:play grandstack
```

Рис. 4.2. Загрузка учебного набора данных в Neo4j

Она загрузит учебный набор данных в экземпляр Neo4j, который будет использоваться в качестве основы для нашего GraphQL API. Выполнив команду в листинге 4.2, мы сможем исследовать данные, включенные в учебный набор (рис. 4.3).

Листинг 4.2. Команда получения схемы графа в Neo4j

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

Рис. 4.3. Схема графа учебного набора данных

Как видите, у нас есть четыре метки узлов – `Business`, `Review`, `Category` и `User`, – связанные тремя типами отношений: `IN_CATEGORY` (связывает компании с категориями), `REVIEWS` (связывает отзывы с компаниями) и `WROTE` (связывает пользователей с отзывами, которые они написали). Также можно просмотреть свойства узлов, как показано в листинге 4.3.

Листинг 4.3. Исследование свойств узлов, хранящихся в Neo4j

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

Эта команда отобразит таблицу с именами свойств, их типами и узлами, в которых они присутствуют:

"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

Мы воспользуемся данной таблицей ниже, когда будем определять типы GraphQL, описывающие этот граф.

Приложение Node.js

Теперь, загрузив учебный набор данных в Neo4j, настроим новый проект Node.js для нашего GraphQL API:

```
npm init -y
```

Также установим необходимые зависимости:

- `@neo4j/graphql` – пакет, упрощающий взаимодействия между GraphQL и Neo4j. Библиотека Neo4j GraphQL преобразует запросы GraphQL в один запрос Cypher, избавляя от необходимости писать запросы в функциях разрешения GraphQL и выполнять пакетные запросы. Он также предоставляет поддержку языка запросов Cypher в GraphQL через директиву схемы `@cypher`;
- `apollo-server` – Apollo Server – сервер GraphQL с открытым исходным кодом, обслуживающий любую схему GraphQL, созданную с помощью graphql.js, включая использование возможностей библиотеки Neo4j GraphQL. Также поддерживает работу со многими различными веб-серверами на платформе Node.js или Express.js;
- `graphql` – эталонная реализация GraphQL.js для JavaScript, требуется двумя другими нашими зависимостями: `@neo4j/graphql` и `apollo-server`. На момент написания этих строк пакет `@neo4j/graphql` был совместим с версией `graphql` 15.x; поэтому мы установим последнюю версию v15.x;
- `neo4j-driver` – клиентские драйверы Neo4j, которые позволяют подключаться к экземпляру Neo4j, как локальному, так и удаленному, и выполнять запросы Cypher по протоколу Bolt. Драйверы Neo4j доступны для многих языков, и мы будем использовать драйвер Neo4j для JavaScript:

```
npm i @neo4j/graphql graphql neo4j-driver apollo-server
```

Теперь создадим новый файл index.js и добавим в него следующий код (листинг 4.4).

Листинг 4.4. index.js: начальный код реализации GraphQL API

```
const { ApolloServer } = require("apollo-server");
const neo4j = require("neo4j-driver");
const { Neo4jGraphQL } = require("@neo4j/graphql");

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

const typeDefs = /* GraphQL */ ``;

const neoSchema = new Neo4jGraphQL({ typeDefs, driver });

neoSchema.getSchema().then((schema) => {
  const server = new ApolloServer({
    schema
  });
```

Импорт зависимостей

Создание соединения с базой данных Neo4j

Эта строка служит заполнителем для определений типов GraphQL, которые будут добавлены позже

Передача определений типов GraphQL и соединения с базой данных при создании экземпляра класса Neo4jGraphQL

Сгенерированная схема GraphQL передается в Apollo Server

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

Запуск сервера GraphQL

Это базовая структура кода нашего приложения GraphQL API. Учетные данные, используемые при создании экземпляра драйвера Neo4j, будут зависеть от того, используете ли вы Neo4j Desktop, Neo4j Sandbox или Neo4j Aura, а также от первоначально выбранного пароля. Обязательно настройте учетные данные для вашего конкретного экземпляра Neo4j.

Если попытаться запустить приложение GraphQL API прямо сейчас, то мы получим сообщение об ошибке, уведомляющее об отсутствии определений типов GraphQL. Мы должны добавить типы, определяющие GraphQL API, поэтому на следующем шаге заполним определения типов GraphQL.

4.3.2. Генерирование схемы GraphQL из определений типов

Согласно типичному подходу разработки приложений GraphQL, описанному выше, определения типов GraphQL задают спецификацию API. В нашем случае мы знаем, какими данными будет оперировать приложение (учебный набор данных уже загружен в Neo4j), поэтому можно обратиться к таблице свойств узлов, показанной выше, и применить простое правило создания определений типов GraphQL: метки узлов становятся типами, а их свойства – полями. Дополнительно необходимо определить поля отношений. Давайте сначала рассмотрим полные определения типов (листинг 4.5), а затем займемся определением полей отношений.

Листинг 4.5. index.js: определения типов GraphQL

```
const typeDefs = /* GraphQL */ `

  type Business {
    businessId: ID!
    name: String!
    city: String!
    state: String!
    address: String!
    location: Point!
    reviews: [Review!]! @relationship(type: "REVIEWS", direction: IN)
    categories: [Category!]!
      @relationship(type: "IN_CATEGORY", direction: OUT)
  }

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

  type Review {
```

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

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

Директива @relationship

В графовой модели свойств, используемой в Neo4j, каждая связь имеет направление и тип. Для отражения типов и направлений связей в схемах GraphQL используются директивы, в частности директива `@relationship`. Директивы похожи на аннотации и добавляются в определения типов GraphQL. Директивы могут принимать необязательные списки именованных аргументов и представляют встроенный механизм расширения GraphQL, указывающий на некоторую пользовательскую логику, которую должен использовать сервер GraphQL.

В директиве `@relationship`, определяющей поля отношений, аргумент `type` сообщает тип отношения, а аргумент `direction` – направление отношения. Помимо директив, используемых в схемах, есть также директивы, применяемые в запросах GraphQL для реализации определенного поведения. Мы увидим несколько примеров директив в запросах, когда будем изучать управление состоянием клиента с помощью Apollo Client в нашем приложении React.

А теперь запустим API нашего приложения:

```
node index.js
```

В выводе вы должны увидеть адрес, на котором API принимает запросы, – в данном случае это порт 4000 на хосте `localhost`:

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

Введите адрес **http://localhost:4000** в веб-браузере, и он должен открыть целевую страницу Apollo Studio. Щелкните на значке **Schema** (Схема) в верхнем левом углу, чтобы увидеть полностью сгенерированный API (рис. 4.4). Потратьте пару минут, чтобы прочитать описания полей запроса. Вы заметите, что к типам добавились аргументы, управляющие упорядочиванием, разбиением на страницы и фильтрацией. Также можете заглянуть на вкладки **Reference** (Справочник) и **SDL**, чтобы увидеть полное определение схемы GraphQL на языке SDL (Schema Definition Language), сгенерированное из наших первоначальных определений типов.

Рис. 4.4. Описание сгенерированного API в Apollo Studio

4.4. Основы запросов GraphQL

Теперь, имея сервер GraphQL на базе Apollo Server и библиотеку Neo4j GraphQL, можно попробовать послать запрос в API с помощью Apollo Studio. На вкладке **Schema** (Схема) в Apollo Studio можно увидеть доступные точки входа в API (на языке GraphQL каждое поле типа `Query` является точкой входа в API): `Business`, `User`, `Review` и `Category` – по одной для каждого типа. Для начала запросим список названий всех компаний, как показано в листинге 4.6.

Листинг 4.6. Запрос GraphQL, возвращающий список названий всех компаний

```
{
  businesses {
    name
  }
}
```

Если запустить этот запрос в Apollo Studio, то в ответ появится список с названиями компаний:

```
{
  "data": {
    "businesses": [
      {
        "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"
      }
    ]
  }
}
```

Отлично! Эти данные были получены из нашего экземпляра Neo4j, и нам даже не пришлось писать функции разрешения!

Давайте включим отладочное журналирование для библиотеки Neo4j GraphQL, чтобы посмотреть, как сгенерированные запросы Cypher отправляются в базу данных. Для этого нужно установить переменную окружения `DEBUG`. Остановим наш сервер GraphQL, нажав **Ctrl-C** в терминале, установим переменную окружения `DEBUG` и снова запустим GraphQL API:

```
DEBUG=@neo4j/graphql:* node index.js
```

Затем повторим наш запрос GraphQL и проверим вывод в терминале. Мы увидим сгенерированный запрос Cypher, как показано в листинге 4.7.

Листинг 4.7. Сгенерированный запрос Cypher

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

В запрос GraphQL можно добавить дополнительные поля, и они добавятся в сгенерированный запрос Cypher. В результате мы вновь получим только запрошенные поля. Например, добавим в запрос GraphQL поля `address` и `name` (листинг 4.8).

Листинг 4.8. Запрос GraphQL, возвращающий названия и адреса компаний

```
{
  businesses {
    name
    address
  }
}
```

Запрос Cypher, получившийся в результате преобразования запроса GraphQL, теперь также будет включать поля `address` и `name` (листинг 4.9).

Листинг 4.9. Сгенерированный запрос Cypher включает поля address и name

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

Наконец, исследовав результаты выполнения запроса GraphQL, можно увидеть, что для каждой компании теперь дополнительно выводится ее адрес:

```
{
  "data": {
    "businesses": [
      {
        "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"
      }
    ]
  }
}
```

Далее попробуем воспользоваться некоторыми функциями сгенерированного GraphQL API.

4.5. Упорядочение и разбиение на страницы

Каждое поле в запросе может принимать входной параметр с объектом `options`. В нем можно указать значения `limit` и `sort`, чтобы организовать упорядочивание и разбиение на страницы. Запрос в листинге 4.10 выбирает первые три компании из списка, упорядоченного полю `name`.

Листинг 4.10. Первая попытка включить аргументы sort и limit в запрос к GraphQL API

```
{
  businesses(options: { limit: 3, sort: { name: ASC } }) {
    name
  }
}
```

Перечисления, определяющие порядок сортировки, создаются для всех типов и включают варианты сортировки по возрастанию и убыванию для каждого поля. Запрос в листинге 4.10 возвращает компании, упорядоченные по названиям, как показано в листинге 4.11.

Листинг 4.11. Результат упорядочения и разбиения на страницы

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

```
    ]
  }
}
```

Переключив внимание на терминал, можно увидеть запрос Cypher, сгенерированный на основе нашего запроса GraphQL; теперь он включает операторы `ORDER BY` и `LIMIT`, соответствующие аргументам `orderBy` и `first` в запросе GraphQL, как показано в листинге 4.12. Это означает, что упорядочивание и ограничение выполняются в базе данных, а не на стороне клиента, поэтому запрос к базе данных возвращает только необходимые данные.

Листинг 4.12. Сгенерированный запрос Cypher, включающий операторы ORDER BY и LIMIT

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

Обратите внимание, что в этом запросе передается имя параметра `$first`, а не число 3. Использование параметра – важный аспект, потому что применение параметров гарантирует невозможность со стороны пользователя внедрить потенциально вредоносный код Cypher в сгенерированный запрос, а кроме того, план сгенерированного запроса может быть повторно использован для повышения производительности. Чтобы выполнить этот запрос в Neo4j Browser, сначала задайте значение параметра `first` с помощью команды `:param`:

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

4.6. Вложенные запросы

Язык запросов GraphQL позволяет выражать обход типов в графе, а библиотека Neo4j GraphQL способна генерировать эквивалентные запросы Cypher для произвольных запросов GraphQL, включая вложенные. Теперь мы перейдем от компаний к категориям, как показано в листинге 4.13.

Листинг 4.13. Запрос GraphQL, включающий вложенный подзапрос

```
{
  businesses(options: { limit: 3, sort: { name: ASC } }) {
    name
    categories {
      name
    }
  }
}
```

Как показывает результат, каждая компания связана с одной или несколькими категориями:

```
{
  "data": {
    "businesses": [
      {
        "name": "Ducky's Car Wash",
        "categories": [
          {
            "name": "Car Wash"
          }
        ]
      },
      {
        "name": "Hanabi",
        "categories": [
          {
            "name": "Ramen"
          },
          {
            "name": "Restaurant"
          }
        ]
      },
      {
        "name": "Imagine Nation Brewing",
        "categories": [
          {
            "name": "Beer"
          },
          {
            "name": "Brewery"
          }
        ]
      }
    ]
  }
}
```

4.7. Фильтрация

Фильтрация осуществляется добавлением аргумента `where` к соответствующим типам GraphQL в запросе, определяющего критерии фильтрации. Полный список критериев фильтрации можно увидеть в документации по адресу **https://neo4j.com/docs/graphql-manual/current/filtering/**.

4.7.1. *Аргумент where*

В листинге 4.14 показан пример использования аргумента `where` для поиска компаний с названиями, содержащими слово Brew (пивоваренный).

Листинг 4.14. Запрос GraphQL, возвращающий список компаний с названиями, содержащими слово «Brew»

```
{
  businesses(where: { name_CONTAINS: "Brew" }) {
    name
    address
  }
}
```

Теперь в результатах возвращаются только пивоваренные компании, соответствующие критериям фильтрации, т. е. в названиях которых содержится слово «Brew»:

```
{
  "data": {
    "businesses": [
      {
        "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. Вложенные фильтры

Для фильтрации по значениям вложенных полей можно использовать вложенные аргументы `where`. В листинге 4.15 показан пример поиска компаний, содержащих в названии слово «Brew» и имеющих хотя бы один отзыв с рейтингом не ниже 4,75.

Листинг 4.15. Запрос GraphQL с вложенным фильтром

```
{
  businesses(
    where: { name_CONTAINS: "Brew", reviews_SOME: { stars_GTE: 4.75 } }
  ) {
    name
    address
  }
}
```

Судя по результатам, в нашем наборе данных имеются две компании, соответствующие этим критериям:

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

4.7.3. Логические операторы: AND, OR

Фильтры можно объединять друг с другом с помощью логических операторов `OR` и `AND`. Например, можно попробовать отыскать компании, входящие в категорию `Coffee` (кофе) или `Breakfast` (завтрак), используя оператор `OR` в аргументе фильтра, как показано в листинге 4.16.

Листинг 4.16. Запрос GraphQL с логическим оператором OR в фильтре

```
{
  businesses(
    where: {
      OR: [
        { categories_SOME: { name: "Coffee" } }
        { categories_SOME: { name: "Breakfast" } }
      ]
    }
  ) {
    name
    address
    categories {
      name
    }
  }
}
```

Этот запрос вернет компании, входящие в категорию `Coffee` или `Breakfast`:

```
{
  "data": {
    "businesses": [
      {
        "name": "Market on Front",
```

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

4.7.4. Фильтрация выборки

Фильтры также можно применять к выбранному набору данных. Например, в листинге 4.17 показано, как отыскать компании, предлагающие кофе или завтрак, в отзывах к которым содержится словосочетание `breakfast sandwich` (бутерброды на завтрак).

Листинг 4.17. Запрос GraphQL с фильтром по полученной выборке

```
{
  businesses(
    where: {
      OR: [
        { categories_SOME: { name: "Coffee" } }
        { categories_SOME: { name: "Breakfast" } }
      ]
    }
  ) {
    name
    address
    reviews(where: { text_CONTAINS: "breakfast sandwich" }) {
      stars
      text
    }
  }
}
```

Поскольку фильтр применяется к отобранным отзывам `reviews`, в результатах по-прежнему присутствуют компании, не имеющие отзывов со словосочетанием `breakfast sandwich`, но отображаются только отзывы, содержащие эту фразу:

```
{
  "data": {
    "businesses": [
      {
        "name": "Market on Front",
        "address": "201 E Front St",
        "reviews": []
      },
      {
        "name": "Ninja Mike's",
        "address": "200 W Pine St",
        "reviews": [
          {
            "stars": 4,
            "text": "Best breakfast sandwich at the Farmer's Market."
          }
        ]
      },
      {
        "name": "Zootown Brew",
        "address": "121 W Broadway St",
        "reviews": []
      }
    ]
  }
}
```

4.8. Работа с датой/временем

Neo4j поддерживает собственные типы для представления даты/времени в свойствах узлов и отношений: `Date`, `DateTime` и `LocalDateTime`. Эти типы можно использовать в схемах GraphQL.

4.8.1. *Использование типа Date в запросах*

В нашей схеме тип `Date` используется в определении типа `Review`. Значением типа `Date` является строка вида `yyyy-mm-dd`, но в базе данных эти данные хранятся в своем внутреннем формате и для них поддерживаются типичные операции с датами. Давайте запросим три самых последних отзыва (листинг 4.18).

Листинг 4.18. Запрос GraphQL, использующий поле даты

```
{
  reviews(options: { limit: 3, sort: { date: DESC } }) {
    stars
    date
    business {
      name
    }
  }
}
```

Поскольку поле даты указано для выборки, оно появится в результатах:

```
{
  "data": {
    "reviews": [
      {
        "stars": 3,
        "date": "2018-09-10",
        "business": {
          "name": "Imagine Nation Brewing"
        }
      },
      {
        "stars": 5,
        "date": "2018-08-11",
        "business": {
          "name": "Zootown Brew"
        }
      },
      {
        "stars": 4,
        "date": "2018-03-24",
        "business": {
          "name": "Market on Front"
        }
```

```
      }
    ]
  }
}
```

4.8.2. Фильтры по полям с типами Date и DateTime

Поля, хранящие дату/время, тоже включаются в сгенерированные перечисления фильтрации, что позволяет фильтровать объекты на основе дат и их диапазонов. Запрос в листинге 4.19 отыскивает отзывы, созданные до 1 января 2017 года.

Листинг 4.19. Запрос GraphQL с фильтром по дате

```
{
  reviews(
    where: { date_LTE: "2017-01-01" }
    options: { limit: 3, sort: { date: DESC } }
  ) {
    stars
    date
    business {
      name
    }
  }
}
```

А вот его результаты, упорядоченные по возрастанию значения в поле `date`:

```
{
  "data": {
    "reviews": [
      {
        "stars": 5,
        "date": "2016-11-21",
        "business": {
          "name": "Hanabi"
        }
      },
      {
        "stars": 5,
        "date": "2016-07-14",
        "business": {
          "name": "KettleHouse Brewing Co."
        }
      },
      {
        "stars": 5,
        "date": "2016-03-04",
        "business": {
          "name": "Ducky's Car Wash"
        }
```

```
      }
    ]
  }
}
```

4.9. Работа с пространственными данными

В настоящее время Neo4j поддерживает тип точки в пространстве, способный представлять двумерные (например, широту и долготу) и трехмерные (например, *x*, *y*, *z* или широту, долготу, высоту) координаты точки, как в географической (широта и долгота), так и в декартовой системе координат. Библиотека Neo4j GraphQL поддерживает два пространственных типа: `Point` – для точек в географической системе координат и `CartesianPoint` – для точек в декартовой системе координат. Более подробную информацию о поддержке пространственных данных в библиотеке Neo4j GraphQL можно найти в документации по адресу: **http://mng.bz/qYKA**.

4.9.1. *Выборка данных типа Point*

Поля с типом `Point` – это объектные поля в схеме GraphQL. Давайте для примера извлечем поля широты и долготы наших компаний, добавив их в список выборки, как показано в листинге 4.20.

Листинг 4.20. Запрос GraphQL, извлекающий поля типа Point

```
{
  businesses(options: { limit: 3, sort: { name: ASC } }) {
    name
    location {
      latitude
      longitude
    }
  }
}
```

Теперь в результатах можно увидеть долготу и широту местоположения каждой компании:

```
{
  "data": {
    "businesses": [
      {
        "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. Фильтрация по расстояниям

Запрашивая данные, представляющие точки в пространстве, часто бывает нужно оставить в результате только те из них, которые расположены недалеко от заданной точки. Например, какие компании находятся в пределах 1,5 км от меня? Такую фильтрацию можно реализовать, используя сгенерированный аргумент в фильтре `where`, как показано в листинге 4.21.

Листинг 4.21. Запрос GraphQL с фильтром по расстоянию

```
{
  businesses(
    where: {
      location_LT: {
        distance: 3500
        point: { latitude: 37.563675, longitude: -122.322243 }
      }
    }
  ) {
    name
    address
    city
    state
  }
}
```

Для точек, представленных в географической системе координат (широта и долгота), расстояние измеряется в метрах:

```
{
  "data": {
    "businesses": [
```

```
      {
        "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. Добавление своей логики в GraphQL API

До сих пор мы рассматривали простые запросы, созданные библиотекой Neo4j GraphQL. Но иногда бывает нужно добавить свою логику в API, например чтобы вычислить самую популярную компанию или порекомендовать какую-то компанию пользователям. Существует два варианта добавления своей логики в API с помощью библиотеки Neo4j GraphQL: с помощью директивы схемы `@cypher` и путем реализации своих функций разрешения.

4.10.1. *Директива @cypher*

Библиотека Neo4j GraphQL предоставляет доступ к Cypher посредством директивы `@cypher`. Добавьте к полю в своей схеме директиву `@cypher`, чтобы отобразить результаты запроса в это поле. Директива `@cypher` принимает оператор с одним аргументом `statement`, представляющим запрос Cypher. Параметры передаются в этот запрос во время выполнения, включая `this`, представляющий текущий узел, а также любые аргументы уровня поля, объявленные в определении типа GraphQL.

> **ПРИМЕЧАНИЕ.** Директива `@cypher` и другие функции библиотеки Neo4j GraphQL требуют подключать плагин APOC из стандартной библиотеки. Не забудьте установить этот плагин, как описано в разделе «Настройка проекта» данной главы.

Вычисляемые скалярные поля

Директиву `@cypher` можно использовать для определения скалярного поля и таким способом создать вычисляемое поле в нашей схеме. В листинге 4.22 показано,

как добавить поле `averageStars` к типу `Business`, вычисляющее среднее количество звезд из всех отзывов о текущей компании с использованием переменной `this`.

Листинг 4.22. index.js: добавление поля `averageStars`

```
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!]! @relationship(type: "REVIEWS", direction: IN)
  categories: [Category!]!
    @relationship(type: "IN_CATEGORY", direction: OUT)
}
```

Перед опробованием этого запроса нужно перезапустить сервер GraphQL, потому что изменилось определение типа:

```
DEBUG=@neo4j/graphql:* node index.js
```

Теперь добавим поле `averageStars` в запрос GraphQL (листинг 4.23).

Листинг 4.23. Запрос GraphQL, включающий поле `averageStars`

```
{
  businesses {
    name
    averageStars
  }
}
```

В результатах теперь можно увидеть новое вычисляемое поле `averageStars`:

```
{
  "data": {
    "Business": [
      {
        "name": "Hanabi",
        "averageStars": 5
      },
      {
        "name": "Zootown Brew",
        "averageStars": 5
      },
      {
```

```
        "name": "Ninja Mike's",
        "averageStars": 4.5
      }
    ]
  }
}
```

Если посмотреть на сгенерированный запрос Cypher в терминале, то можно заметить, что он включает запрос из директивы `@cypher` как подзапрос. Таким образом, в базу данных по-прежнему передается единственный запрос, но включающий нашу собственную логику!

Вычисляемые поля с объектами и массивами

Директиву `@cypher` также можно использовать для вычисления полей, представляющих объекты и массивы. Давайте добавим поле `recommended` в тип `Business` и используем простой запрос Cypher, отыскивающий похожие компании, для которых писали отзывы другие пользователи (листинг 4.24). Например, если пользователю нравится компания `Market on Front`, мы можем порекомендовать ему другие компании, для которых писали отзывы иные пользователи, оставившие отзывы для `Market on Front`.

Листинг 4.24. Запрос Cypher, выбирающий компании для рекомендации

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

Этот запрос Cypher можно добавить в схему GraphQL, включив его в директиву `@cypher` для поля `recommended` в определении типа `Business` (листинг 4.25).

```
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]-(u:User)
      MATCH (u)-[: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!]! @relationship(type: "REVIEWS", direction: IN)
  categories: [Category!]!
    @relationship(type: "IN_CATEGORY", direction: OUT)
}
```

Мы также определили аргумент `first` для поля, который передается в запрос Cypher, указанный в директиве `@cypher`, и действует как ограничение на количество компаний, возвращаемых в рекомендации.

Настраиваемые поля запроса верхнего уровня

Еще один полезный способ использования директивы `@cypher` – настройка извлечения или изменения поля. Давайте посмотрим, как добавить поддержку полнотекстовых запросов для поиска компаний (листинг 4.26). Приложения часто используют полнотекстовый поиск, например для исправления опечаток во вводе пользователя с помощью нечеткого сопоставления. В Neo4j можно применять полнотекстовый поиск, предварительно создав полнотекстовый индекс.

Листинг 4.26. Cypher: создание полнотекстового индекса

```
CREATE FULLTEXT INDEX businessNameIndex FOR (b:Business) ON EACH [b.name]
```

Чтобы проверить работу этого индекса, наберем `libary`, допустив опечатку, и добавим символ ~, обозначающий нечеткое сопоставление, чтобы отыскать искомое (листинг 4.27).

Листинг 4.27. Cypher: запрос полнотекстового индекса

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

А можно ли включить этот полнотекстовый поиск с нечетким сопоставлением в GraphQL API? Да, можно. Для этого нужно создать поле типа `Query`, например с именем `fuzzyBusinessByName`, определяющее запрос, который принимает искомую строку и выполняет поиск компаний, как показано в листинге 4.28.

Листинг 4.28. index.js: добавление поля типа `Query`

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

И снова, поскольку изменения коснулись определений типов, необходимо перезапустить сервер GraphQL:

```
DEBUG=@neo4j/graphql:* node index.js
```

Заглянув на вкладку **Schema** (Схема) в Apollo Studio, можно увидеть новое поле типа `Query` с именем `fuzzyBusinessByName`. С его помощью можно искать названия компаний, используя нечеткое сопоставление, как показано в листинге 4.29.

Листинг 4.29. Запрос GraphQL, использующий новый запрос `Query`

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

Поскольку мы используем полнотекстовый поиск, то, даже допустив ошибку в названии `libary`, мы все равно найдем желаемые результаты:

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

Директива `@cypher` открывает широкие возможности для добавления собственной логики и расширения возможностей GraphQL API. Директиву `@cypher` можно также использовать для целей авторизации, получения доступа к таким значениям, как токены авторизации, из объекта запроса. Этот способ авторизации мы обсудим в следующей главе, когда будем исследовать разные варианты добавления авторизации в API. Более полную информацию о директиве `@cypher` можно найти в документации: **http://mng.bz/7yom**.

4.10.2. *Реализация собственных функций разрешения*

Директива `@cypher` позволяет добавлять свою логику, но иногда может потребоваться реализовать свои функции разрешения с логикой, которую невозможно выразить на языке Cypher, например чтобы получить данные из другой системы или применить некоторые нестандартные правила проверки. В таких случаях можно написать свою функцию разрешения, добавить ее в схему GraphQL и вызывать для разрешения некоторого поля в обход запросов Cypher, генерируемых библиотекой Neo4j GraphQL.

Давайте представим, например, что есть некоторая внешняя система, которую можно использовать для определения времени ожидания в очереди на получение заказа в компаниях. Для этого добавим дополнительное поле `waitTime` в тип `Business` и реализуем соответствующую функцию разрешения, обращающуюся к этой внешней системе.

Сначала добавим поле в схему, используя директиву `@ignore`, чтобы исключить поле из сгенерированного запроса Cypher, как показано в листинге 4.30. Таким способом мы сообщаем библиотеке Neo4j GraphQL, что за разрешение этого поля отвечает специальная функция и его не следует извлекать из базы данных.

Листинг 4.30. index.js: добавление поля waitTime

```
type Business {
  businessId: ID!
  waitTime: Int! @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!]! @relationship(type: "REVIEWS", direction: IN)
  categories: [Category!]! @relationship(type: "IN_CATEGORY", direction: OUT)
}
```

Затем добавим карту функций разрешения и нашу функцию, как показано в листинге 4.31. Прежде нам не нужно было создавать эту карту, потому что все функции разрешения автоматически генерировала библиотека Neo4j GraphQL. В данном примере наша функция просто выбирает случайное значение, но мы могли бы реализовать здесь любую логику, вычисляющую значение waitTime, например выполняющую запрос к стороннему API.

Листинг 4.31. index.js: создание карты функций разрешения

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

Далее добавим эту карту в параметры конструктора Neo4jGraphQL, как показано в листинге 4.32.

Листинг 4.32. index.js: генерирование схемы GraphQL

```
const neoSchema = new Neo4jGraphQL({typeDefs, resolvers, driver})
```

Теперь перезапустим приложение GraphQL API, поскольку изменения коснулись определений типов в схеме:

```
DEBUG=@neo4j/graphql:* node index.js
```

Если после перезапуска Apollo Studio проверить схему для типа Business, то можно увидеть наше новое поле waitTime. Давайте попробуем найти рестораны (листинг 4.33) и посмотрим, какое время ожидания у каждого из них, добавив поле waitTime в выборку.

Листинг 4.33. Запрос GraphQL, использующий поле с нашей функцией разрешения

```
{
  businesses(where: { categories_SOME: { name: "Restaurant" } }) {
    name
    waitTime
  }
}
```

Теперь в результатах можно увидеть время ожидания. Ваши результаты, конечно, будут отличаться от приведенных здесь, потому что значение выбирается случайно:

```
{
  "data": {
    "businesses": [
      {
        "name": "Ninja Mike's",
        "waitTime": 30
      },
      {
        "name": "Market on Front",
        "waitTime": 5
      },
      {
        "name": "Hanabi",
        "waitTime": 45
      }
    ]
  }
}
```

4.11. Определение схемы GraphQL в существующей базе данных

Как правило, приступая к разработке нового приложения, у нас нет готовой базы данных, и мы следуем парадигме разработки, согласно которой сначала определяются типы GraphQL. Однако иногда может иметься готовая база данных Neo4j, заполненная информацией. В таких случаях можно сгенерировать определения типов GraphQL на основе существующей базы данных и затем передать в библиотеку Neo4j GraphQL для создания GraphQL API. Сделать это можно с помощью пакета `@neo4j/introspector`.

Сначала следует установить пакет `@neo4j/introspector`:

```
npm i @neo4j/introspector
```

Этот сценарий Node.js подключится к базе данных Neo4j, проанализирует определения типов GraphQL, описывающие данные, как показано в листинге 4.34, и запишет эти определения в файл с именем schema.graphql.

Листинг 4.34. intropect.js: импорт определений типов GraphQL

```
const { toGraphQLTypeDefs } = require("@neo4j/introspector");
const neo4j = require("neo4j-driver");
const fs = require("fs");
const driver = neo4j.driver(
  "neo4j://localhost:7687",
  neo4j.auth.basic("neo4j", "letmein")
);
const sessionFactory = () =>
  driver.session({ defaultAccessMode: neo4j.session.READ });

// Определение асинхронной функции для использования инструкции async/await
async function main() {
  const typeDefs = await toGraphQLTypeDefs(sessionFactory);
  fs.writeFileSync("schema.graphql", typeDefs);
  await driver.close();
}

main();
```

Затем загрузим файл schema.graphql и передадим определения типов конструктору `Neo4jGraphQL` (листинг 4.35).

Листинг 4.35. Загрузка определений типов GraphQL из schema.graphql

```
// Загрузить определения типов GraphQL из файла schema.graphql
const typeDefs = fs
  .readFileSync(path.join(__dirname, "schema.graphql"))
  .toString("utf-8");
```

До сих пор все наши запросы GraphQL выполнялись с использованием Apollo Studio, инструмента, который отлично подходит для тестирования и разработки, но главная наша цель – создать приложение, обращающееся к GraphQL API. В следующих нескольких главах мы приступим к созданию пользовательского интерфейса нашего приложения с использованием React и Apollo Client. Попутно познакомимся с другими концепциями GraphQL, такими как мутации, фрагменты, типы интерфейса, и многими другими!

4.12. Упражнения

1. Используя Apollo Studio, сконструируйте запрос к GraphQL API, созданному в этой главе, возвращающий:
   - список пользователей, оставивших отзыв о компании `Hanabi`;
   - любые отзывы, в которых есть слово «comfortable» (удобный) и соответствующие им компании;
   - список пользователей, не давших ни одной компании 5 звезд.

2. Добавьте в тип `Category` поле с директивой `@cypher`, вычисляющее количество компаний в каждой категории. Сколько компаний относится к категории `Coffee`?
3. Создайте экземпляр Neo4j Sandbox по адресу **https://sandbox.neo4j.com**, выбрав любой из предварительно заполненных наборов данных. Используя пакет `@neo4j/introspector`, импортируйте определения типов и создайте GraphQL API для этого экземпляра, не написав в определениях типов ни строчки вручную. Выясните, какие данные можно запрашивать с помощью полученного GraphQL API.

Загляните в репозиторий книги на GitHub, где приводятся решения упражнений: **http://mng.bz/mOYP**.

Итоги

- К типичным проблемам, возникающим при создании GraphQL API, относятся: проблема $n + 1$ запросов, дублирование схемы и большой объем типового кода для выборки данных.
- Интеграция GraphQL с базой данных, например с помощью библиотеки Neo4j GraphQL, может помочь смягчить эти проблемы путем автоматического создания: запросов к базе данных на основе запросов GraphQL, схемы базы данных и GraphQL API из определений типов GraphQL.
- Библиотека Neo4j GraphQL упрощает создание GraphQL API, поддерживаемых базой данных Neo4j, предлагая поддержку автоматически создаваемых функций разрешения для выборки данных и фильтрации, упорядочивания и разбиения на страницы.
- Добавить свою логику в GraphQL API можно с помощью директивы `@cypher` с логикой обработки полей или путем реализации своих функций разрешения.
- Если имеется готовая база данных Neo4j, то с помощью пакета `@neo4j/introspector` из нее можно извлечь определения типов GraphQL и на их основе создать GraphQL API.

Часть II

Создание пользовательского интерфейса

В первой части все наше внимание было сосредоточено на серверной части приложения. В ней мы познакомились с графовой базой данных Neo4j и реализовали GraphQL API с использованием библиотеки Neo4j GraphQL. Теперь пришло время заняться клиентским приложением с пользовательским интерфейсом на основе React.

В главе 5 мы рассмотрим структуру React и основные идеи, которые пригодятся, когда мы приступим к созданию клиентского приложения на основе React. Затем, в главе 6, добавим выборку данных и управление состоянием клиента с помощью React и GraphQL по мере извлечения данных из GraphQL API, созданного в предыдущих главах. К концу второй части у нас появится первая действующая версия приложения обзора компаний, и мы будем готовы изучить возможности добавления авторизации и развертывания в части III.

Глава 5

Создание пользовательского интерфейса с помощью React

В этой главе:

- обзор основных понятий React;
- начало работы с React и инструмент Create React App;
- работа с состоянием в приложении React с использованием обработчиков React Hooks.

До сих пор все свое внимание мы уделяли внутренним аспектам приложения: созданию GraphQL API и операциям с базой данных. Теперь пришло время обратить наши взоры на пользовательский интерфейс. В главе 1 мы кратко познакомились с React и рассмотрели минимальный фрагмент кода компонента React. В данной главе мы вернемся к этой библиотеке и начнем создавать приложение, играющее роль клиента нашего GraphQL API. Оно будет искать компании и отображать результаты в браузере. Конечно, невозможно вместить в одну главу все, что нужно знать о React, поэтому главной ее целью будем считать не исчерпывающее введение в React, а описание фундаментальных понятий, необходимых для создания простого приложения. Для простоты мы будем использовать инструмент командной строки Create React App, а за более полной информацией о React я советую обращаться к документации и учебным пособиям, доступным по адресу: **https://reactjs.org/**.

В этой главе мы создадим основу приложения React, используя Create React App для управления инструментами сборки и конфигурацией. Затем мы добавим в эту основу компоненты, реализующие поиск компаний по категориям и просмотр результатов. На первом этапе данные будут жестко «зашиты» в приложении, но потом, в главе 6, мы добавим логику выборки данных из GraphQL API, созданного в предыдущих главах с помощью Apollo Client (рис. 5.1).

Приступим!

Рис. 5.1. В этой главе обсуждается создание приложения React, играющего роль клиента нашего GraphQL API

5.1. Обзор React

React – это обычная библиотека на JavaScript для создания пользовательских интерфейсов. Ее можно использовать для создания пользовательских веб- (ReactDOM), мобильных (React Native) и других интерфейсов, таких как виртуальная реальность (React VR). React использует идею *компонентов* для инкапсуляции данных и логики. Компоненты можно повторно использовать, объединять друг с другом и создавать из них сложные пользовательские интерфейсы, но все они предоставляют стандартную абстракцию, помогающую разработчикам рассуждать о своих приложениях. Для полноценного использования React необходимо понимать следующие важные концепции: JSX, элементы React, свойства, состояние, подключаемые обработчики (hooks) и иерархия компонентов.

5.1.1. *JSX и элементы React*

Роль основных строительных блоков в React играют *элементы*. Не путайте элементы с компонентами – компоненты состоят из элементов React. Элемент – это нечто, что можно видеть в пользовательском интерфейсе. Например, рассмотрим простой элемент React в листинге 5.1.

Листинг 5.1. Простой элемент React, определенный с помощью JSX

```
const element = <h1>Welcome to GRANDstack</h1>;
```

На первый взгляд, это фрагмент HTML, но с некоторыми намеками на JavaScript. По сути, это пример JSX, используемого для создания элементов React.

> **ПРИМЕЧАНИЕ.** JSX не требуется для работы с React, но настоятельно рекомендуется, и альтернативы в этой книге рассматриваться не будут.

JSX можно рассматривать как комбинацию HTML и JavaScript. В JSX можно использовать выражения на JavaScript, заключая их в фигурные скобки. Например, чтобы персонализировать приветствие «Добро пожаловать в GRANDstack», можно использовать переменную JavaScript с именем пользователя (листинг 5.2).

Листинг 5.2. Использование выражений на JavaScript в JSX

```
const name = "Bob Loblaw";
const element = <h1>Welcome to GRANDstack, {name}!</h1>
```

На этапе сборки код JSX компилируется в код на JavaScript и использует функцию `React.createElement()` для создания элементов React, которые в основной своей массе являются объектами JavaScript, отображаемыми в DOM.

Элементы React играют важную роль, помогая библиотеке React поддерживать так называемую виртуальную объектную модель документа (Document Object Model, DOM) – представление DOM, позволяющее библиотеке применять обновления для приведения DOM в желаемое состояние. Это означает, что при появлении изменений перерисовывается не все дерево DOM, а только изменившиеся его части.

5.1.2. *Компоненты React*

React позволяет создавать пользовательский интерфейс, используя небольшие составные части, называемые *компонентами*. Компоненты – это, по сути, функции, принимающие входные данные (свойства) и возвращающие элементы React, из которых формируется пользовательский интерфейс. Пример одного из таких компонентов показан в листинге 5.3.

> **ПРИМЕЧАНИЕ.** Мы будем использовать только функциональные компоненты React. В документации можно увидеть ссылки на так называемые компоненты-классы; однако с появлением в React класса `Hooks` потребность в компонентах-классах отпала.

Листинг 5.3. Простой компонент React

```
function Greeting(props) {
  return <h1>Welcome to GRANDstack, {props.name}</h1>;
}
```

Компоненты используют данные двух типов: *свойства* (props) и *состояние* (state). Свойство не может изменяться; для изменения значения в компоненте, вызывающего повторное отображение на экране, следует использовать состояния.

5.1.3. *Иерархия компонентов*

Компоненты React могут состоять из других компонентов. Это позволяет инкапсулировать и повторно использовать логические компоненты при создании пользовательского интерфейса (листинг 5.4).

Листинг 5.4. Составной компонент React

```
function Greeting(props) {
  return <h1>Welcome to GRANDstack, {props.name}</h1>;
}

function Popup() {
  const name = "Bob Loblaw";
  return <Greeting name={name} />
}
```

5.2. Create React App

Create React App – это инструмент командной строки, предназначенный для создания приложений React. Он объединяет инструменты сборки, не требует первоначальной настройки, предлагает самый простой способ начать работу с React, автоматически настраивая webpack, Babel, ESLint и другие инструменты, и позволяет разработчикам начать писать приложения React без необходимости устанавливать и настраивать инструменты сборки. Узнать больше о Create React App можно на сайте **https://create-react-app.dev/**.

5.2.1. *Создание приложения React с помощью Create React App*

Давайте создадим приложение React, используя Create React App, и поместим его в каталог рядом с нашим GraphQL API. Напишем для начала несколько основных функций для нашего приложения обзора компаний и начнем с поиска компаний. Первая версия нашего приложения React должна позволять пользователям искать компании по категориям и отображать сведения о них. На этом этапе мы жестко «зашьем» данные в объект JavaScript, а в следующей главе подключим приложение React к GraphQL API и будем извлекать данные оттуда. Чтобы начать работу с Create React App, выполните следующую команду в терминале, находясь в каталоге рядом с нашим GraphQL API:

```
npx create-react-app web-react --use-npm
```

Команда `npx` добавлена в `npm` в версии 5.2.0 и может использоваться для запуска пакетов и команд `npm`. Одна из замечательных особенностей `npx` – способность автоматически загружать пакет, если он не был установлен локально, что гарантирует использование последней версии.

До сих пор мы использовали `npm`, но Create React App по умолчанию применяет диспетчер пакетов `yarn`, поэтому мы будем добавлять флаг `--use-npm` при вызове команды `create-react-app`. После выполнения этой команды в терминале должно появиться сообщение об успешном создании нового проекта React и некоторые

полезные команды, которые мы можем использовать, чтобы начать работу с проектом[1]:

```
Успех! Создано приложение web-react в /Users/lyonwj/business-reviews/web-react
Внутри этого каталога можно выполнить некоторые команды:

  npm start
    Чтобы запустить сервер разработки.

  npm run build
    Чтобы собрать приложение в статические файлы для промышленной эксплуатации.

  npm test
    Чтобы запустить тесты.

  npm run eject
    Чтобы удалить этот инструмент и скопировать зависимости, конфигурационные
    файлы и сценарии в каталог приложения. После этого у вас не будет
    возможности вернуться назад!

Мы предлагаем начать с команд:

  cd web-react
  npm start

Удачной работы!
```

Давайте посмотрим, что создал Create React App:

```
.
├ README.md
├ package-lock.json
├ package.json
├ public
| ├ favicon.ico
| ├ index.html
| ├ logo192.png
| ├ logo512.png
| ├ manifest.json
| └ robots.txt
└ src
  ├ App.css
  ├ App.js
  ├ App.test.js
  ├ index.css
  ├ index.js
  ├ logo.svg
  ├ serviceWorker.js
  └ setupTests.js
└ node_modules
  ├ ...
```

[1] Сообщение, конечно же, выводится на английском языке, но я посчитал нужным перевести его на русский язык, так как в нем есть немного полезной информации. – *Прим. перев.*

Файл README.md содержит документацию с описанием только что созданного приложения React и Create React App. В каталоге `node_modules` находятся все зависимости нашего приложения, установленные автоматически. Внутри каталога `public` можно найти статический контент, обслуживаемый из корня приложения после его запуска. В каталоге src находится код на JavaScript и файлы CSS, определяющие каркас приложения React. Давайте первым рассмотрим файл package.json (листинг 5.5), где перечислены сценарии и подключенные зависимости.

Листинг 5.5. package.json

```
{
  "name": "web-react",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.15.1",
    "@testing-library/react": "^11.2.7",
    "@testing-library/user-event": "^12.8.3",
    "react": "^17.0.2",
    "react-dom": "^17.0.2",
    "react-scripts": "4.0.3",
    "web-vitals": "^1.1.2"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Здесь можно видеть подключенные зависимости нашего приложения: библиотека React, а также пакет под названием `react-scripts`. Пакет `react-scripts` отве-

чает за запуск, сборку и тестирование приложения, как можно видеть в разделе `scripts`. Продолжим и запустим приложение:

```
cd web-react
npm start
```

Команда `npm start` создает сборку для разработки и запускает локальный веб-сервер, обслуживающий приложение React. За окружением разработки ведется постоянное наблюдение, поэтому любые изменения в файлах с исходным кодом немедленно вызывают перезапуск приложения. Благодаря этому отпадает необходимость перезапускать веб-сервер после внесения изменений в код, чтобы увидеть их в действии:

```
Компиляция завершилась успешно!

Теперь можно подключиться к приложению web-react в браузере.

  Адрес на локальном компьютере: http://localhost:3000
  Адрес в вашей сети:            http://192.168.1.3:3000

Обратите внимание, что сборка для разработки не оптимизирована.
Чтобы создать сборку для эксплуатации, выполните команду npm run build.
```

Если приложение запустилось успешно, то появится сообщение, подсказывающее, как открыть приложение в веб-браузере (рис. 5.2).

Рис. 5.2. Начальная версия нашего приложения React в веб-браузере

Давайте откроем файл src/App.js и посмотрим на код начальной версии нашего приложения (листинг 5.6).

Листинг 5.6. src/App.js: код начальной версии приложения

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Этот файл экспортирует компонент `App`, но где он используется? Если открыть src/index.js, то можно увидеть, как используется компонент `App` (листинг 5.7). Он передается в вызов `ReactDOM.render` и сообщает классу `ReactDOM`, что компонент `App` должен отображаться в HTML-элементе с идентификатором `root`.

Листинг 5.7. src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// Чтобы оценить работу приложения, вызовите функцию журналирования
// результатов (например: reportWebVitals(console.log))
```

```
// или передайте их конечной точке analytics. Дополнительную информацию
// ищите по адресу: https://bit.ly/CRA-vitals
reportWebVitals();
```

Давайте дополним файл src/App.js, как показано в листинге 5.8. Для начала создадим простую форму с раскрывающимся списком для поиска компаний по категориям.

Листинг 5.8. src/App.js: добавление образцов данных и простой формы

```
const businesses = [
  {
    businessId: "b1",
    name: "San Mateo Public Library",
    address: "55 W 3rd Ave",
    category: "Library",
  },
  {
    businessId: "b2",
    name: "Ducky's Car Wash",
    address: "716 N San Mateo Dr",
    category: "Car Wash",
  },
  {
    businessId: "b3",
    name: "Hanabi",
    address: "723 California Dr",
    category: "Restaurant",
  },
];

function App() {
  return (
    <div>
      <h1>Business Search</h1>
      <form>
        <label>
          Select Business Category:
          <select value="All">
            <option value="All">All</option>
            <option value="Library">Library</option>
            <option value="Restaurant">Restaurant</option>
            <option value="Car Wash">Car Wash</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
```

На данный момент данные определены в виде массива в коде на JavaScript

Наш компонент React находится на вершине иерархии, и ему не передаются никакие свойства, поэтому он не принимает никаких аргументов

```
      </form>

      <h2>Results</h2>
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Address</th>
            <th>Category</th>
          </tr>
        </thead>
        <tbody>
          {businesses.map((b, i) => (
            <tr key={i}>
              <td>{b.name}</td>
              <td>{b.address}</td>
              <td>{b.category}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}

export default App;
```

Массив компаний отображается в таблицу, где для каждой компании создается отдельная строка

Пока что мы определили данные о компаниях в форме массива, но позже будем извлекать данные из нашего GraphQL API. Первоначально все результаты отображаются в простой HTML-таблице (рис. 5.3).

Business Search

Select Business Category: All Submit

Results

Name	Address	Category
San Mateo Public Library	55 W 3rd Ave	Library
Ducky's Car Wash	716 N San Mateo Dr	Car Wash
Hanabi	723 California Dr	Restaurant

Рис. 5.3. Вид приложения React после дополнения файла src/App.js

Приложение исправно выводит таблицу, но форма пока не работает. Мы можем выбрать категорию в раскрывающемся списке, но это никак не влияет на содержимое таблицы. Давайте вернемся к нашему приложению и добавим фильтрацию результатов в соответствии с выбранной категорией. Но для это-

го нам нужно поближе познакомиться с понятиями состояния и свойств! Поскольку мы используем исключительно функциональные компоненты React, то для работы с состоянием мы должны использовать подключаемые обработчики React Hooks.

5.3. Состояние и подключаемые обработчики React Hooks

Обработчики React Hooks появились в версии React 16.8 и дают возможность работы с состоянием (и другими составляющими React), позволяя использовать функциональные компоненты React вместо классов. Возможно, вам доводилось видеть компоненты-классы React, включающие вызовы функции `setState`, методов управления жизненным циклом и конструкторов. Обработчики React Hooks избавляют от всего этого; благодаря им мы можем управлять состоянием с помощью отдельных вызовов функций.

Далее мы познакомимся с обработчиками на практике, расширив наше приложение React и добавив в него функции фильтрации, которые помогут сформировать результаты для таблицы со списком компаний. Попутно мы увидим, как использовать State React Hook для управления состоянием в компоненте.

Определим новый компонент React, который будет отвечать за отображение нашей таблицы, и назовем его `BusinessResults`. Для этого создадим новый файл с именем BusinessResults.js в том же каталоге, где находится App.js, как показано в листинге 5.9.

Листинг 5.9. src/BusinessResults.js

```
function BusinessResults(props) {
  const { businesses } = props;

  return (
    <div>
      <h2>Results</h2>
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Address</th>
            <th>Category</th>
          </tr>
        </thead>
        <tbody>
          {businesses.map((b, i) => (
            <tr key={i}>
              <td>{b.name}</td>
              <td>{b.address}</td>
              <td>{b.category}</td>
```

В аргумент `props` компоненту передаются свойства

Аргумент `props` содержит данные для отображения в таблице

```
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}

export default BusinessResults;
```

Переместим таблицу с результатами в этот файл src/BusinessResults.js, а компании для отображения компонентом передадим в аргументе `props`. Вместо всех компаний компонент будет отображать в таблице только данные, переданные в аргументе `props`. Далее в нашем компоненте `App` импортируем этот новый компонент `BusinessResults` и передадим ему наш массив данных через свойство компонента, как показано в листинге 5.10.

Листинг 5.10. src/App.js: использование компонента `BusinessResults`

```
import BusinessResults from "./BusinessResults";

const businesses = [
  {
    businessId: "b1",
    name: "San Mateo Public Library",
    address: "55 W 3rd Ave",
    category: "Library",
  },
  {
    businessId: "b2",
    name: "Ducky's Car Wash",
    address: "716 N San Mateo Dr",
    category: "Car Wash",
  },
  {
    businessId: "b3",
    name: "Hanabi",
    address: "723 California Dr",
    category: "Restaurant",
  },
];

function App() {
  return (
    <div>
      <h1>Business Search</h1>
      <form>
        <label>
```

Импорт компонента `BusinessResults`

```
          Select Business Category:
          <select value="All">
            <option value="All">All</option>
            <option value="Library">Library</option>
            <option value="Restaurant">Restaurant</option>
            <option value="Car Wash">Car Wash</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>

      <BusinessResults businesses={businesses} />
    </div>
  );
}
export default App;
```

Передача компоненту `BusinessResults` **массива компаний в качестве свойства**

Мы импортировали новый компонент `BusinessResults` и передали ему массив компаний. Теперь `BusinessResults` может отобразить результаты, а компоненту `App` достаточно разрешить пользователю выбирать категорию для поиска.

Однако после внесения этого изменения приложение выглядит в веб-браузере точно так же, как и прежде, и форма выбора по-прежнему не работает. Давайте добавим функциональности раскрывающемуся списку, как показано в листинге 5.11!

Листинг 5.11. src/App.js: использование переменной состояния

```
import React, { useState } from "react";
import BusinessResults from "./BusinessResults";

const businesses = [
  {
    businessId: "b1",
    name: "San Mateo Public Library",
    address: "55 W 3rd Ave",
    category: "Library",
  },
  {
    businessId: "b2",
    name: "Ducky's Car Wash",
    address: "716 N San Mateo Dr",
    category: "Car Wash",
  },
  {
    businessId: "b3",
    name: "Hanabi",
    address: "723 California Dr",
```

Импорт обработчика `useState`

```
    category: "Restaurant",
  },
];

function App() {

  const [selectedCategory, setSelectedCategory] = useState("All");

  return (
    <div>
      <h1>Business Search</h1>
        <form>
          <label>
            Select Business Category:
            <select
              value={selectedCategory}
              onChange={(event) => setSelectedCategory(event.target.value)}
            >
              <option value="All">All</option>
              <option value="Library">Library</option>
              <option value="Restaurant">Restaurant</option>
              <option value="Car Wash">Car Wash</option>
            </select>
          </label>
          <input type="submit" value="Submit" />
        </form>

        <BusinessResults
          businesses={
            selectedCategory === "All"
              ? businesses
              : businesses.filter((b) => {
                return b.category === selectedCategory;
          })
        }
      />
    </div>
  );
}

export default App;
```

Вызов обработчика `useState`**, чтобы создать новую переменную состояния и функцию для обновления ее значения**

Передача значения, выбранного в раскрывающемся списке, в нашу новую переменную состояния

Обновление значения переменной состояния при выборе пользователем нового пункта в раскрывающемся списке

Фильтрация результатов, передаваемых компоненту `BusinessResults`**, на основе выбранной категории**

Прежде всего импортируем обработчик `useState` и используем его для создания новой переменной состояния `selectedCategory`. Вызов `useState` также возвращает функцию (которую мы назвали `setSelectedCategory`); она используется для обновления значения `selectedCategory`. Затем переменной состояния присваивается значение, выбранное в раскрывающемся списке, с помощью функции

`setSelectedCategory`. Теперь пользователь может выбрать значение в форме и увидеть в таблице только компании, относящиеся к выбранной категории (рис. 5.4).

Рис. 5.4. Вид приложения React после добавления поддержки состояния и фильтрации

Теперь, после создания простого приложения React, следующим нашим шагом станет добавление функции извлечения данных из нашего GraphQL API. Мы сделаем это в следующей главе, где задействуем обработчики Apollo Client React Hooks и попутно познакомимся с дополнительными возможностями React!

5.4. Упражнения

1. Переместите логику поиска в новый компонент с именем `BusinessSearch` и используйте его из компонента `App`.
2. В дополнение к фильтрации по категории добавьте фильтрацию по городу. Для этого нужно добавить город в набор исходных данных и включить его вывод в таблице с результатами.
3. Как бы вы реализовали поиск по нескольким категориям? Измените набор данных так, чтобы каждая компания принадлежала нескольким категориям. Измените форму, реализовав возможность выбора нескольких категорий. Обновите логику фильтрации для передачи правильных результатов поиска компоненту `BusinessResults`.

Итоги

- React – это библиотека JavaScript для создания пользовательских интерфейсов. Она использует идею компонентов для инкапсуляции логики. Компоненты можно комбинировать для создания сложных пользовательских интерфейсов.
- JSX – это синтаксис, используемый для создания элементов React. Он позволяет применять HTML-подобный синтаксис для создания кода, определяющего пользовательский интерфейс.
- Компоненты React используют данные в двух формах: свойств и состояния. Свойства (или реквизиты) – это неизменяемые данные, которые передаются компонентам как часть одностороннего потока данных React.

Состояния – это локальные и частные данные для конкретного компонента, при изменении они вызывают повторное отображение дерева компонентов.

- Create React App – это инструмент командной строки для создания приложений React. Он объединяет инструменты сборки и не требует первоначальной настройки.
- React Hooks – это подключаемые обработчики, позволяющие работать с состоянием внутри компонента, сохраняя при этом компоненты как функции.

Глава 6

Клиент GraphQL

В этой главе:

- подключение приложения React к конечной точке GraphQL с помощью Apollo Client;
- кеширование и обновление данных на стороне клиента с помощью Apollo Client;
- обновление данных в приложении с помощью мутаций GraphQL;
- использование Apollo Client для управления состоянием клиента React.

В предыдущей главе мы с помощью Create React App создали приложение React, позволяющее искать компании по категориям. В качестве источника данных в этом приложении использовался жестко «вшитый» в код объект JavaScript, поэтому приложение имело ограниченную функциональность. В этой главе мы посмотрим, как подключить приложение React к GraphQL API, созданному в предыдущих главах, и познакомимся с новым инструментом в наборе GraphQL: Apollo Client.

Apollo Client – это библиотека JavaScript, предназначенная для управления данными и позволяющая разработчикам управлять локальными и удаленными данными с помощью GraphQL. Она применяется для извлечения, кеширования и изменения данных приложения и предлагает возможность интеграции с внешним интерфейсом, включая React, для обновления пользовательского интерфейса синхронно с изменением данных.

Мы также используем React Hooks API для передачи данных в приложение React из нашего GraphQL API, выполняя запросы GraphQL с помощью Apollo Client. После этого займемся исследованием операций мутации (изменения) GraphQL, с помощью которых можно изменять данные в GraphQL API, и посмотрим, как обрабатывать события изменения данных в приложении. Наконец, мы познакомимся с приемами использования Apollo Client для управления локальным состоянием приложения React путем добавления локальных полей в GraphQL API.

Приступим!

6.1. Apollo Client

Apollo Client – это больше, чем просто библиотека, которая отправляет и принимает графовые данные. Вот небольшая выдержка из документации Apollo Client:

> *Apollo Client – это комплексная библиотека управления состоянием для JavaScript, которая позволяет управлять локальными и удаленными данными с помощью GraphQL. Ее можно использовать для извлечения, кеширования и изменения данных приложения при поддержке автоматического обновления пользовательского интерфейса. ... Apollo Client помогает писать компактный, предсказуемый и декларативный код, соответствующий современным требованиям. Основная библиотека @apollo/client имеет встроенную интеграцию с React, а обширное сообщество Apollo реализовало интеграцию с другими популярными фреймворками представления.*
>
> – **https://www.apollographql.com/docs/react/**

Мы воспользуемся всеми этими возможностями Apollo Client для добавления в наше приложение сначала логики выборки данных, а затем для управления локальными данными состояния.

6.1.1. *Добавление Apollo Client в приложение React*

Поскольку мы используем библиотеку React, то сосредоточимся на интеграции с Apollo Client, характерной именно для React. Сначала установим Apollo Client с помощью `npm`, создадим экземпляр Apollo Client, подключим его к GraphQL API, а затем начнем выполнять запросы для получения данных, используя обработчик `useQuery`, предоставляемый Apollo Client.

Поскольку мы собираемся обращаться к нашему GraphQL API, не забудьте запустить базу данных Neo4j и приложение GraphQL API из предыдущих глав. Если этого не сделать, то в терминале появятся сообщения об ошибках, указывающие на недоступность конечной точки GraphQL.

Установка Apollo Client

На момент написания этих строк самой свежей была версия Apollo Client 3.5.5 и большинство инструментов, необходимых для добавления поддержки GraphQL в приложение React, входили в состав одного пакета. Предыдущие версии Apollo Client поставлялись отдельно от React Hooks; однако теперь интеграция с React включена по умолчанию.

Откройте терминал, перейдите в каталог `web-react` и выполните следующую команду, чтобы установить Apollo Client. Также необходимо установить зависимость `graphql.js`. Мы используем самую свежую версию Apollo Client, имевшуюся на момент написания этих строк, т. е. v3.5.5:

```
npm install @apollo/client graphql
```

После установки Apollo Client можно создать экземпляр Apollo Client и начать выполнять запросы GraphQL. Сначала мы посмотрим, как это сделать в общем случае, а затем добавим эту поддержку в наше приложение React.

Создание экземпляра Apollo Client

Чтобы создать экземпляр Apollo Client, нужно передать конструктору URI GraphQL API для подключения, а также кеш, как показано в листинге 6.1. Чаще всего в роли кеша используется `InMemoryCache`.

Листинг 6.1. Создание экземпляра Apolo Client

```
import { ApolloClient, InMemoryCache } from "@apollo/client";

const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache(),
});
```

Полученный экземпляр клиента можно использовать для выполнения операций GraphQL.

Выполнение запросов с помощью Apollo Client

Взгляните на листинг 6.2, где показано, как выполнить запрос GraphQL с помощью клиентского API. В нашем приложении React мы будем использовать в основном React Hooks API клиента Apollo Client, поэтому этот код не будет использоваться в нашем приложении.

Листинг 6.2. Выполнение запросов с помощью Apollo Client

```
import { ApolloClient, InMemoryCache, gql } from "@apollo/client";

const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache(),
});

client
  .query({
    query: gql`
      {
        businesses {
          name
      }
    }
  `
})
.then(result => console.log(result));
```

Обратите внимание, что наш запрос GraphQL заключен в литеральный тег `gql`. Это сделано для преобразования строки запроса GraphQL в стандартное абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) GraphQL, понятное клиен-

там GraphQL. Здесь мы выполняем операцию `query`, чтобы получить список компаний, состоящий только из их названий, и выводим его в консоль.

Результат выполнения этого минимального примера Apollo Client показан на рис. 6.1. Наш экземпляр Apollo Client отправляет запрос на сервер GraphQL. Тот, в свою очередь, возвращает данные, которые затем сохраняются в кеше Apollo Client. Последующие запросы тех же данных будут извлекаться из кеша без отправки запроса серверу GraphQL. Ниже в этой главе я расскажу, как напрямую работать с кешем Apollo Client.

Рис. 6.1. Минимальный пример Apollo Client

Теперь, познакомившись с основными возможностями Apollo Client, посмотрим, как использовать их в нашем приложении React.

Внедрение Apollo Client в иерархию компонентов

Первое, что нужно сделать, – внедрить экземпляр клиента в иерархию компонентов React, сделав его доступным для каждого из них. Для этого внесем несколько изменений в файл web-react/src/index.js, созданный с помощью Create React App (листинг 6.3).

Листинг 6.3. web-react/src/index.js: создание экземпляра Apollo Client в приложении

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
import {
  ApolloClient,
  InMemoryCache,
  ApolloProvider,
} from "@apollo/client";

const client = new ApolloClient({   ← Создание экземпляра Apollo Client
  uri: "http://localhost:4000",
  cache: new InMemoryCache(),
});

ReactDOM.render(
```

```
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>,
  document.getElementById("root")
);

// Чтобы оценить работу приложения, вызовите функцию журналирования
// результатов (например: reportWebVitals(console.log))
// или передайте их конечной точке analytics. Дополнительную информацию
// ищите по адресу: https://bit.ly/CRA-vitals
reportWebVitals();
```

Использование компонента `ApolloProvider` для внедрения экземпляра клиента в иерархию компонентов React

После создания экземпляра Apollo Client, подключенного к нашему GraphQL API, мы заключаем наш компонент `App` в компонент `ApolloProvider` и передаем последнему экземпляр клиента в виде реквизита. Это позволит любому компоненту в нашем приложении получить доступ к экземпляру клиента и выполнять операции GraphQL через React Hooks API (рис. 6.2).

Рис. 6.2. Внедрение экземпляра Apollo Client в иерархию компонентов React

6.1.2. *Обработчики Apollo Client*

К средствам интеграции Apollo Client и React, кроме всего прочего, относятся обработчики React Hooks. Обработчик `useQuery` – это основной метод выполнения запросов GraphQL. Чтобы узнать, как использовать обработчик `useQuery`, обновим наш компонент `App` и реализуем поиск данных в GraphQL API вместо жестко «вшитого» массива (листинг 6.4).

Листинг 6.4. web-react/src/App.js: добавление запроса к GraphQL API

```
import React, { useState } from "react";
import BusinessResults from "./BusinessResults";
```

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

const GET_BUSINESSES_QUERY = gql`
  {
    businesses {
      businessId
      name
      address
      categories {
        name
      }
    }
  }
`;

function App() {
  const [selectedCategory, setSelectedCategory] = useState("All");

  const { loading, error, data } = useQuery(GET_BUSINESSES_QUERY);

  if (error) return <p>Error</p>;
  if (loading) return <p>Loading...</p>;

  return (
    <div>
      <h1>Business Search</h1>
      <form>
        <label>
          Select Business Category:
          <select
            value={selectedCategory}
            onChange={(event) => setSelectedCategory(event.target.value)}
          >
            <option value="All">All</option>
            <option value="Library">Library</option>
            <option value="Restaurant">Restaurant</option>
            <option value="Car Wash">Car Wash</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>

      <BusinessResults businesses={data.businesses} />
    </div>
  );
}

export default App;
```

Импорт обработчика `useQuery`

Определение запроса GraphQL для поиска компаний

Обработчик `useQuery` **предоставляет различные состояния жизненного цикла операции GraphQL**

Передача ответа GraphQL компоненту `BusinessResults`

Прежде всего необходимо импортировать обработчик `useQuery` и литеральный тег `gql`. Затем мы определяем запрос GraphQL для поиска компаний и возвращаем данные для отображения в нашей таблице результатов. Далее передаем запрос GraphQL обработчику `useQuery`, который возвращает объекты состояния, позволяющие проверить выполнение операции GraphQL: `loading` (загрузка), `error` (ошибка) и `data` (данные). Пока запрос загружается, можно показать индикатор, сообщающий пользователю, что идет извлечение данных. Если запрос вернет ошибку, то можно вывести текст ошибки. Наконец, после заполнения объекта данных можно быть уверенными, что запрос GraphQL успешно выполнен, и передать эти данные в виде свойства компоненту `BusinessResults`, отобразить их в таблице с результатами (рис. 6.3).

Рис. 6.3. Потоки данных в приложении React, перемещаемых с помощью обработчиков Apollo Client

Нам также потребуется внести небольшие коррективы в компонент `BusinessResults`, так как теперь каждая компания может принадлежать нескольким категориям (листинг 6.5).

Листинг 6.5. web-react/src/BusinessResults.js: отображение категорий

```
function BusinessResults(props) {
  const { businesses } = props;

  return (
```

```
    <div>
      <h2>Results</h2>
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Address</th>
            <th>Category</th>
          </tr>
        </thead>
        <tbody>
          {businesses.map((b, i) => (
            <tr key={i}>
              <td>{b.name}</td>
              <td>{b.address}</td>
              <td>
                {b.categories.reduce(
                  (acc, c, i) => acc + (i === 0 ? " " : ", ") + c.name,
                  ""
                )}
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}

export default BusinessResults;
```

Функция `reduce` используется для создания общего строкового представления списка категорий

Если теперь открыть наше приложение в браузере, то должна появиться заполненная таблица с результатами. На этот раз данные поступают из GraphQL API (рис. 6.4).

Business Search

Select Business Category: All Submit

Results

Name	Address	Category
Missoula Public Library	301 E Main St	Library
Ninja Mike's	200 W Pine St	Restaurant, Breakfast
KettleHouse Brewing Co.	313 N 1st St W	Beer, Brewery
Imagine Nation Brewing	1151 W Broadway St	Beer, Brewery
Market on Front	201 E Front St	Coffee, Restaurant, Cafe, Deli, Breakfast
Hanabi	723 California Dr	Restaurant, Ramen
Zootown Brew	121 W Broadway St	Coffee
Ducky's Car Wash	716 N San Mateo Dr	Car Wash
Neo4j	111 E 5th Ave	Graph Database

Рис. 6.4. Вид нашего приложения React после подключения к GraphQL API

Конечно, наше приложение пока недостаточно функционально, потому что оно просто показывает все компании. Далее нам нужно добавить фильтрацию по категории, выбранной пользователем. Для этого передадим выбранную категорию как переменную GraphQL.

6.1.3. *Переменные GraphQL*

Переменные GraphQL позволяют передавать динамические аргументы в операции GraphQL. Давайте изменим web-react/src/App.js, чтобы реализовать поиск компаний, которые соответствуют категории, выбранной пользователем, передав эту категорию в переменной GraphQL. Воспользуемся поддержкой фильтрации, описанной в главе 4, и, в частности, аргументом `where` (листинг 6.6).

Листинг 6.6. web-react/src/App.js: применение переменных GraphQL

```
import React, { useState } from "react";
import BusinessResults from "./BusinessResults";

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

const GET_BUSINESSES_QUERY = gql`
  query BusinessesByCategory($selectedCategory: String!) {
    businesses(
      where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
    ) {
      businessId
      name
      address
      categories {
        name
      }
    }
  }
`;

function App() {
  const [selectedCategory, setSelectedCategory] = useState("");

  const { loading, error, data } = useQuery(GET_BUSINESSES_QUERY, {
    variables: { selectedCategory },
  });

  if (error) return <p>Error</p>;
  if (loading) return <p>Loading...</p>;

  return (
    <div>
```

```
      <h1>Business Search</h1>
      <form>
        <label>
          Select Business Category:
          <select
            value={selectedCategory}
            onChange={(event) => setSelectedCategory(event.target.value)}
          >
            <option value="">All</option>
            <option value="Library">Library</option>
            <option value="Restaurant">Restaurant</option>
            <option value="Car Wash">Car Wash</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>

      <BusinessResults businesses={data.businesses} />
    </div>
  );
}

export default App;
```

При работе с переменными GraphQL сначала нужно заменить статическое значение на `$selectedCategory`. Затем объявить `$selectedCategory` как одну из переменных, принимаемых запросом. Потом передать значение для `$selectedCategory` в вызов `useQuery`. Теперь результаты поиска будут автоматически обновляться при выборе категории и содержать только компании из этой категории (рис. 6.5).

Рис. 6.5. Результат реализации фильтрации по категориям с помощью GraphQL

6.1.4. *Фрагменты GraphQL*

До сих пор при создании запроса выборки мы перечисляли все поля, в том числе вложенные. Но в приложениях очень часто разные компоненты выбирают одни и те же наборы (или подмножества) полей. Эти наборы можно оформить в виде *фрагментов GraphQL* и повторно использовать в запросах. Для этого сначала нужно объявить фрагмент, назначив ему имя и тип, как показано в листинге 6.7.

Листинг 6.7. Объявление фрагмента GraphQL

```
fragment businessDetails on Business {
    businessId
    name
    address
    categories {
        name
    }
}
```

Здесь мы определили фрагмент с именем `businessDetails`, который можно использовать для выбора полей типа `Business` и который включает все поля, необходимые для отображения в нашей таблице результатов. Чтобы использовать этот фрагмент, нужно добавить его имя в запрос, добавив перед ним многоточие ..., как показано в листинге 6.8.

Листинг 6.8. Использование фрагмента в запросе GraphQL

```
query BusinessesByCategory($selectedCategory: String!) {
  businesses(
    where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
  ) {
    ...businessDetails
    }
  }
}
```

Этот запрос возвращает те же результаты, что и предыдущий, но теперь у нас есть возможность повторно использовать фрагмент `businessDetails` в других запросах.

Использование фрагментов с Apollo Client

Чтобы использовать фрагменты с Apollo Client, нужно объявить их в отдельных переменных и включить в запросы GraphQL, используя заполнители в шаблоне. Это позволяет хранить фрагменты и применять их несколькими компонентами. Если позднее понадобится изменить поля в выборке, это придется сделать только в одном месте – в определении фрагмента, и все запросы, использующие этот фрагмент, автоматически обновятся.

Итак, объявим наш фрагмент `businessDetails` в переменной `BUSINESS_DETAILS_FRAGMENT`, а затем включим в запрос GraphQL, используя заполнитель в шаблоне, как показано в листинге 6.9.

Листинг 6.9. web-react/src/App.js: использование фрагмента GraphQL

```
...

const BUSINESS_DETAILS_FRAGMENT = gql`
```

```
  fragment businessDetails on Business {
    businessId
    name
    address
    categories {
      name
    }
  }
`;

const GET_BUSINESSES_QUERY = gql`
  query BusinessesByCategory($selectedCategory: String!) {
    businesses(
      where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
    ) {
      ...businessDetails
    }
  }

  ${BUSINESS_DETAILS_FRAGMENT}
`;

...
```

6.1.5. *Кеширование в Apollo Client*

Apollo Client сохраняет результаты в нормализованном кеше в памяти. Это означает, что при попытке повторно выполнить тот же запрос GraphQL Apollo Client не отправит его на сервер, а просто прочитает результаты из кеша, что сократит затраты времени на ненужные сетевые запросы и повысит воспринимаемую производительность приложения. Чтобы убедиться, что результаты действительно кешируются, можно открыть инструменты разработчика в браузере и проверить вкладку сети, выбирая различные категории в раскрывающемся списке.

Обновление кешированных результатов

Кеширование способствует повышению производительности, когда данные меняются не очень часто, но как быть, если после выполнения запроса и кеширования результата данные на сервере обновились? Можно ли отказаться от кеширования, чтобы всегда получать самые свежие данные с сервера? К счастью, в Apollo Client есть параметр, управляющий обновлением кешированных результатов. Мы рассмотрим два варианта обновления кешированных данных: опрос и повторную выборку.

Опрос (polling) позволяет синхронизировать результаты с заданным интервалом. В Apollo Client опрос можно включить для каждого запроса, указав значение `pollInterval` в миллисекундах. Например, в листинге 6.10 показано, как установить интервал опроса равным 500 мс.

Листинг 6.10. web-react/src/App.js: установка интервала опроса

```
const { loading, error, data } = useQuery(GET_BUSINESSES_QUERY, {
    variables: { selectedCategory },
    pollInterval: 500
});
```

Вместо обновления результатов через постоянные интервалы времени можно использовать прием *повторной выборки* (refetching), позволяющий явно потребовать обновить данные в кеше, например в ответ на такие действия пользователя, как щелчок на кнопке или отправка формы. Чтобы выполнить повторную выборку в Apollo Client, нужно вызвать функцию `refetch`, возвращаемую обработчиком `useQuery`, как показано в листинге 6.11.

Листинг 6.11. web-react/src/App.js: использование функции refetch

```
import React, { useState } from "react";
import BusinessResults from "./BusinessResults";

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

const GET_BUSINESSES_QUERY = gql`
  query BusinessesByCategory($selectedCategory: String!) {
    businesses(
      where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
    ) {
      businessId
      name
      address
      categories {
        name
      }
    }
  }
`;

function App() {
  const [selectedCategory, setSelectedCategory] = useState("");

  const { loading, error, data, refetch } = useQuery(
    GET_BUSINESSES_QUERY,
    {
      variables: { selectedCategory },
    }
  );

  if (error) return <p>Error</p>;
```

Функция `refetch` **возвращается обработчиком** `useQuery`

```
  if (loading) return <p>Loading...</p>;

  return (
    <div>
      <h1>Business Search</h1>
      <form>
        <label>
          Select Business Category:
          <select
            value={selectedCategory}
            onChange={(event) => setSelectedCategory(event.target.value)}
          >
            <option value="">All</option>
            <option value="Library">Library</option>
            <option value="Restaurant">Restaurant</option>
            <option value="Car Wash">Car Wash</option>
          </select>
        </label>
        <input type="button" value="Refetch" onClick={() => refetch()} />
      </form>

      <BusinessResults businesses={data.businesses} />
    </div>
  );
}

export default App;
```

Щелчок на кнопке приводит к вызову `refetch`

Теперь, когда мы готовы обрабатывать изменяющиеся данные, посмотрим, как можно обновлять их с помощью мутаций GraphQL.

6.2. Мутации GraphQL

Мутации GraphQL – это операции, записывающие новые или изменяющие старые данные. Идея мутаций была кратко представлена в главе 2, но до сего момента мы не использовали их. В этом разделе мы исследуем мутации, генерируемые библиотекой Neo4j GraphQL, позволяющие создавать, изменять и удалять узлы и отношения.

6.2.1. *Создание узлов*

Мутация создания генерируется для каждого типа, описанного в наших определениях GraphQL, и отображается в метку узла в Neo4j. Чтобы создать узлы, нужно вызвать соответствующую мутацию создания, передав ей значения свойств нового узла. Обратите внимание, что если какое-то поле определено с восклицательным знаком (!), то оно не может быть пустым и должно быть включено в число аргументов, передаваемых мутации. Давайте для примера добавим в базу новую компанию: Philz Coffee (листинг 6.12).

Листинг 6.12. Создание новой компании с помощью мутации GraphQL

```
mutation {
  createBusinesses(
    input: {
      businessId: "b10"
      name: "Philz Coffee"
      address: "113. S B St"
      city: "San Mateo"
      state: "CA"
      location: { latitude: 37.567109, longitude: -122.323680 }
    }
  ) {
    businesses {
      businessId
      name
      address
      city
    }
    info {
      nodesCreated
    }
  }
}
```

Запуск этой мутации в Apollo Studio создаст новый узел типа `Business` в базе данных:

```
{
  "data": {
    "createBusinesses": {
      "businesses": [
        {
          "businessId": "b10",
          "name": "Philz Coffee",
          "address": "113. S B St",
          "city": "San Mateo"
        }
      ],
      "info": {
        "nodesCreated": 1
      }
    }
  }
}
```

6.2.2. Создание отношений

Для создания отношений в базе данных можно использовать операции обновления, сгенерированные библиотекой Neo4j GraphQL. В листинге 6.13 показано, как связать новый узел Philz Coffee с узлом категории `Coffee`. Для этого передадим мутации идентификатор конкретного узла.

```
mutation {
  updateBusinesses(
    where: { businessId: "b10" }
    connect: { categories: { where: { node: { name: "Coffee" } } } }
  ) {
    businesses {
      name
      categories {
        name
      }
    }
    info {
      relationshipsCreated
    }
  }
}
```

Обратите внимание на аргумент `connect`. Он позволяет создавать отношения между уже существующими узлами. Аналогично можно создать новый узел категории, используя аргумент `create`; однако в данном случае узел категории `Coffee` уже существует в базе данных. Аргументы `connect` и `create` также доступны в мутациях создания узлов и составляют мощную особенность библиотеки Neo4j GraphQL, называемую *вложенными мутациями*. Вкладывая операции создания или связывания, можно выполнить несколько операций записи в одной мутации GraphQL:

```
{
  "data": {
    "updateBusinesses": {
      "businesses": [
        {
          "name": "Philz Coffee",
          "categories": [
            {
              "name": "Coffee"
            }
          ]
        }
      ],
      "info": {
        "relationshipsCreated": 1
```

```
      }
    }
  }
}
```

6.2.3. Изменение и удаление

Давайте представим, что кофейня Philz Coffee переехала по другому адресу, в здание напротив офиса Neo4j, и нам нужно обновить ее адрес в базе данных. Для этого можно использовать мутацию `updateBusinesses`, задав значение в поле `businessId` для ссылки на узел, и затем любые значения для других полей, которые необходимо изменить в аргументе `update` (листинг 6.14).

Листинг 6.14. Мутация GraphQL для изменения адреса компании

```
mutation {
  updateBusinesses(
    where: { businessId: "b10" }
    update: { address: "113 E 5th Ave" }
  ) {
    businesses {
      name
      address
      categories {
        name
      }
    }
  }
}

{
  "data": {
    "updateBusinesses": {
      "businesses": [
        {
          "name": "Philz Coffee",
          "address": "113 E 5th Ave",
          "categories": [
            {
              "name": "Coffee"
            }
          ]
        }
      ]
    }
  }
}
```

Полностью удалить узел из базы данных можно с помощью мутации `deleteBusinesses` (листинг 6.15).

Листинг 6.15. Мутация GraphQL для удаления узла компании

```
mutation {
  deleteBusinesses(where: { businessId: "b10" }) {
    nodesDeleted
  }
}

{
  "data": {
    "deleteBusinesses": {
      "nodesDeleted": 1
    }
  }
}
```

Экспериментируя с этими операциями мутации в Apollo Studio, попробуйте применить методы опроса и повторной выборки, упомянутые в предыдущем разделе, чтобы увидеть, как приложение React реагирует на изменение внутренних данных по мере выполнения мутаций.

6.3. Управление состоянием клиента с помощью GraphQL

Выше мы говорили, что Apollo Client – это комплексная библиотека управления данными, позволяющая работать не только с данными на сервере GraphQL, но и с локальными данными. Локальные данные могут определять состояние приложения React, например пользовательские настройки, которые не требуется отправлять на сервер, потому что они имеют отношение только к клиенту.

Apollo Client позволяет добавлять локальные поля в запросы GraphQL, которые затем могут управляться и кешироваться Apollo Client, что окажет нам большую помощь в поддержке состояния нашего приложения React. Это очень удобно, потому что позволяет использовать один и тот же API для работы и с локальными, и с удаленными данными!

6.3.1. *Локальные поля и реактивные переменные*

В Apollo Client *локальные поля* можно определить и включить в схему GraphQL. Эти поля отсутствуют в схеме на сервере и относятся только к клиентскому приложению. Значения для этих полей вычисляются локально с использованием логики, определяемой нами, такой как сохранение и чтение из localStorage в браузере.

Используя *реактивные переменные*, можно читать и записывать локальные значения, хранящиеся вне GraphQL. Это удобно, когда желательно иметь возможность обновлять значения без привлечения GraphQL (например, в ответ на дей-

ствие пользователя), но читать локальные поля как часть операции выборки данных GraphQL вместе с другой логикой выборки данных с сервера GraphQL. Кроме того, изменение реактивной переменной вызывает обновление любого запроса, использующего ее значение.

Давайте объединим локальное поле с реактивной переменной, чтобы добавить в приложение функцию, выделяющую *компании со звездами*. Добавим кнопку **Star** (Звезда) рядом с каждой компанией в списке результатов, что позволит пользователю отмечать понравившиеся ему компании звездами. Такие компании будут выделяться жирным шрифтом, чтобы пользователь мог видеть, что это одни из его любимых компаний.

Для этого, как показано в листинге 6.16, сначала добавим *политику поля* для локального поля в экземпляр `InMemoryCache`, который используется в Apollo Client. Политика поля определяет порядок вычисления локального поля. Здесь мы добавляем локально поле `isStarred`. Также создается новая реактивная переменная для хранения списка компаний, отмеченных звездой. В этом случае политика для поля `isStarred` проверяет, включена ли данная компания в список отмеченных звездами.

Листинг 6.16. web-react/src/index.js: использование реактивной переменной

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
import {
  ApolloClient,
  InMemoryCache,
  ApolloProvider,
  makeVar,
} from "@apollo/client";

export const starredVar = makeVar([]);

const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache({
    typePolicies: {
      Business: {
        fields: {
          isStarred: {
            read(_, { readField }) {
              return starredVar().includes(readField("businessId"));
            },
          },
        },
      },
```

Импорт функции `makeVar` для создания новой реактивной переменной

Создание новой реактивной переменной установкой начального значения равным пустому массиву

Включение политики поля в аргументы конструктора `InMemoryCache`

Политика поля определяет, как вычисляется значение локального поля с именем `isStarred` в типе `Business`

Возврат значения `true`, если список компаний со звездами включает текущую компанию

```
    },
  }),
});

ReactDOM.render(
  <React.StrictMode>
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  </React.StrictMode>,
  document.getElementById("root")
);

// Чтобы оценить работу приложения, вызовите функцию журналирования
// результатов (например: reportWebVitals(console.log))
// или передайте их конечной точке analytics. Дополнительную информацию
// ищите по адресу: https://bit.ly/CRA-vitals
reportWebVitals();
```

Теперь можно включить поле `isStarred` в запрос GraphQL, как показано в листинге 6.17. Для этого нужно добавить директиву `@client`, чтобы указать, что это поле локальное и не будет получено с сервера GraphQL.

Листинг 6.17. web-react/src/App.js: использование локального поля GraphQL

```
...

const GET_BUSINESSES_QUERY = gql`
  query BusinessesByCategory($selectedCategory: String!) {
    businesses(
      where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
    ) {
      businessId
      name
      address
      categories {
        name
      }
      isStarred @client
    }
  }
`;

...
```

Добавить поле `isStarred` в выборку, указав, что оно локальное, с помощью директивы `@client`

Наконец, нужен какой-то способ, позволяющий изменить реактивную переменную `starredVar`. В листинге 6.18 мы добавляем кнопку **Star** (Звезда) рядом

с каждой компанией. После щелчка на этой кнопке значение `starredVar` обновляется и добавляет выбранную компанию в список отмеченных звездами.

Листинг 6.18. web-react/src/BusinessResults.js: использование реактивной переменной

```
import { starredVar } from "./index";

function BusinessResults(props) {
  const { businesses } = props;
  const starredItems = starredVar();

  return (
    <div>
      <h2>Results</h2>
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Address</th>
            <th>Category</th>
          </tr>
        </thead>
        <tbody>
          {businesses.map((b, i) => (
            <tr key={i}>
              <td>
                <button
                  onClick={() =>
                    starredVar([...starredItems, b.businessId])
                  }
                >
                  Star
                </button>
              </td>
              <td style={b.isStarred ? { fontWeight: "bold" } : null}>
                {b.name}
              </td>
              <td>{b.address}</td>
              <td>
                {b.categories.reduce(
                  (acc, c, i) => acc + (i === 0 ? " " : ", ") + c.name,
                  ""
                )}
              </td>
            </tr>
          ))}
        </tbody>
```

Извлечение значения `starredVar`, чтобы найти все компании, отмеченные звездами

При щелчке добавить `businessId` в список компаний, отмеченных звездами

Если компания отмечена звездой, то вывести ее название жирным шрифтом

```
      </table>
    </div>
  );
}

export default BusinessResults;
```

А поскольку это реактивная переменная, любой активный запрос, зависящий от локального поля `isStarred`, автоматически обновится в пользовательском интерфейсе (рис. 6.6)!

Business Search

Select Business Category: All Submit

Results

Star	Name	Address	Category
Star	Missoula Public Library	301 E Main St	Library
Star	Ninja Mike's	200 W Pine St	Restaurant, Breakfast
Star	KettleHouse Brewing Co.	313 N 1st St W	Beer, Brewery
Star	Imagine Nation Brewing	1151 W Broadway St	Beer, Brewery
Star	Market on Front	201 E Front St	Coffee, Restaurant, Cafe, Deli, Breakfast
Star	**Hanabi**	723 California Dr	Restaurant, Ramen
Star	Zootown Brew	121 W Broadway St	Coffee
Star	**Ducky's Car Wash**	716 N San Mateo Dr	Car Wash
Star	**Neo4j**	111 E 5th Ave	Graph Database

Рис. 6.6. Приложение React с поддержкой локальных полей

Теперь, после знакомства с мутациями, нужно подумать о защите приложения, чтобы никто не мог бесконтрольно добавлять и изменять наши данные. В следующей главе мы посмотрим, как добавить аутентификацию для защиты приложения.

6.4. Упражнения

1. Какие еще фрагменты GraphQL можно было бы использовать в нашем приложении? Напишите несколько фрагментов и попробуйте использовать их в запросах в Apollo Studio. Когда имеет смысл применять несколько фрагментов в одном запросе?
2. Используя мутации GraphQL, создайте отношения между узлами `Business` и `Category`, чтобы можно было добавить компании в дополнительные категории. Например, добавьте только что созданную компанию Philz Coffee в категории `Restaurant` и `Breakfast`. Добавьте в граф свои любимые компании и соответствующие им категории.
3. Превратите кнопку **Star** (Звезда) в переключатель. Если компания уже отмечена звездой, то щелчок на кнопке должен исключать ее из списка отмеченных звездами.

Итоги

- Apollo Client – это библиотека управления данными, позволяющая разработчикам управлять локальными и удаленными данными с помощью GraphQL и поддерживающая возможность интеграции с фреймворками пользовательского интерфейса, такими как React.
- Мутации GraphQL – это операции, позволяющие создавать и изменять данные и генерируемые библиотекой Neo4j GraphQL для каждого типа.
- Apollo Client можно использовать для управления локальным состоянием, добавляя локальные поля в схему GraphQL и определяя соответствующие им политики полей, которые задают порядок чтения, хранения и изменения этих локальных данных.

Часть III

Задачи разработки полного цикла

После создания начальной версии нашего приложения самое время уделить внимание его защите и развертыванию с использованием облачных служб. В главе 7 мы добавим в наш GraphQL API авторизацию и аутентификацию, а также познакомимся со службой Auth0. В главе 8 используем Netlify, AWS Lambda и Neo4j AuraDB для развертывания приложения полного цикла. Наконец, в главе 9, завершающей эту книгу, мы рассмотрим применение абстрактных типов, разбиение на страницы с применением курсора и обработку свойств отношений в GraphQL. После завершения этой части книги мы получим защищенное приложение GraphQL полного цикла, развернутое в облаке.

Глава 7

Добавление авторизации и аутентификации

В этой главе:

- аутентификация и авторизация добавляются с обеих сторон приложения: в GraphQL API и в пользовательский интерфейс React;
- использование веб-токенов JWT (JSON Web Tokens) для кодирования идентификатора пользователя и разрешений;
- определение и применение правил авторизации в схеме GraphQL с применением директивы @auth;
- использование Auth0 в качестве провайдера JWT и Auth0 React SDK для добавления поддержки Auth0 в приложение.

Аутентификация (проверка личности пользователя) и *авторизация* (проверка разрешений на доступ к ресурсам) необходимы для защиты любого приложения. Они выполняют проверку наличия у пользователя необходимых разрешений, защищая доступ к данным и операциям приложения на обеих сторонах, клиента и сервера. На данный момент и наш пользовательский интерфейс, и GraphQL API открыты для всех желающих, и любой сможет получить доступ ко всем функциям и возможностям, включая изменение, создание и удаление данных.

Сама технология GraphQL не имеет встроенных средств авторизации, оставляя выбор решения на усмотрение разработчика. В этой главе мы увидим, как реализовать функции авторизации и аутентификации в приложении, используя JWT, директивы GraphQL и службу Auth0. Для начала мы рассмотрим простейший подход и реализуем авторизацию посредством функций разрешения. Затем познакомимся с директивой схемы `@auth`, реализованной в библиотеке Neo4j GraphQL и позволяющей добавлять правила авторизации в схему. После этого добавим поддержку службы авторизации Auth0 и посмотрим, как можно использовать веб-токены JSON Web Tokens для управления идентификацией пользователей и их разрешениями.

7.1. Авторизация в GraphQL: простейший подход

Для начала рассмотрим простейший подход к добавлению авторизации в GraphQL API (листинг 7.1), используя только один статический токен авторизации. Получив запрос на сервере GraphQL, мы проверим наличие токена в заголовке авторизации запроса, передадим этот токен функции разрешения, где сравним его с предопределенным значением, и отправим соответствующий ответ, только если токен действителен. Обратите внимание, что этот пример предназначен исключительно для иллюстрации идеи и не представляет практической ценности!

Листинг 7.1. api/naive.js: простейшая реализация авторизации в GraphQL

```
const { ApolloServer } = require("apollo-server");

const peopleArray = [
  {
    name: "Bob",
  },
  {
    name: "Lindsey",
  },
];

const typeDefs = /* GraphQL */ `
  type Query {
    people: [Person]
  }

  type Person {
    name: String
  }
`;

const resolvers = {
  Query: {
    people: (obj, args, context, info) => {
      if (
        context &&
        context.headers &&
        context.headers.authorization === "Bearer authorized123"
      ) {
        return peopleArray;
      } else {
        throw new Error("You are not authorized.");
      }
```

Проверка значения токена авторизации

```
    },
  },
};

const server = new ApolloServer({
  resolvers,
  typeDefs,
  context: ({ req }) => {
    return { headers: req.headers };
  },
});

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

Добавление HTTP-заголовков в запрос в объекте `context`

Здесь сервер GraphQL имеет единственную функцию разрешения `Query.people` с логикой проверки токена авторизации, передаваемого через объект контекста. Токен извлекается из заголовка запроса и передается в объект контекста во время обработки запроса (рис. 7.1).

Рис. 7.1. Процесс авторизации в простейшей реализации

Давайте посмотрим, как работает это решение. Запустим сервер GraphQL:

```
node naive.js
```

И в Apollo Studio выполним запрос GraphQL, отыскивающий все объекты `Person` и возвращающий поле `name` каждого:

```
{
  people {
    name
  }
}
```

Запрос был отклонен из-за отсутствия соответствующего токена авторизации, и в результате мы получили сообщение об ошибке «You are not authorized» (Вы не авторизованы). Давайте добавим соответствующий заголовок в запрос GraphQL с токеном авторизации. Для этого в Apollo Studio щелкните на кнопке **Headers** (Заголовки) в левом нижнем углу, выберите **New header** (Создать заголовок) и добавьте в него ключ `Authorization` со значением `Bearer authorized123`:

```
{
  "Authorization": "Bearer authorized123"
}
```

Повторно отправив тот же запрос – на этот раз с токеном авторизации в заголовке, – мы получим ожидаемые результаты:

```
{
  "data": {
    "people": [
      {
        name: "Bob",
      },
      {
        name: "Lindsey",
      },
    ]
  }
}
```

Этот простейший подход демонстрирует несколько важных концепций, например как извлечь заголовок авторизации из запроса и передать его через объект контекста в функцию разрешения, и добавить этот заголовок в запрос в Apollo Studio. Однако данный подход имеет несколько существенных недостатков, весьма нежелательных в реальном приложении.

1. *Мы не проверяем достоверность токена*. Это решение не позволяет убедиться, что пользователь, выполнивший запрос, действительно является тем, за кого себя выдает, и действительно обладает полномочиями, указанными в токене. Мы просто верим им на слово!

2. *Правила авторизации смешаны с логикой выборки данных в функции разрешения.* Может показаться, что это допустимо для простого примера, но представьте, что произойдет, если добавить больше типов и правил авторизации, – нам будет сложно отслеживать и поддерживать их.

Первый недостаток легко исправить, используя JWT с криптографической подписью для кодирования и проверки личности пользователей и их разрешений, выраженных в заголовке авторизации. Для исправления второго недостатка применим директиву схемы `@auth`, реализованную библиотекой Neo4j GraphQL; добавляя декларативные правила авторизации в схему, мы получаем единый источник истины для наших правил авторизации.

7.2. Веб-токены JSON Web Token

JSON Web Token, или просто JWT, – это открытый стандарт создания объектов JSON с криптографической подписью, которые можно использовать для доверенной связи между сторонами. Компактный токен генерируется и подписывается с использованием пары ключей (открытого и закрытого), позволяя убедиться, что он создан стороной, владеющей закрытым ключом, и, следовательно, целостность информации, содержащейся в нем, можно криптографически проверить путем декодирования с использованием открытого ключа, парного закрытому, применявшемуся для подписывания.

Информация (полезная нагрузка), закодированная в JWT, – это серия *утверждений* о чем-то или о ком-то, обычно о пользователе. Вот некоторые стандартные утверждения в JWT:

- `iss` – эмитент токена;
- `exp` – дата истечения действия токена;
- `sub` – субъект, обычно некоторый идентификатор, определяющий пользователя, к которому применяются эти утверждения;
- `aud` – аудитория, часто используется при аутентификации через API.

Также можно добавить дополнительные утверждения, чтобы выразить информацию о пользователе, например его роли в приложении (например, «администратор», «редактор» и т. д.), или более детальные разрешения, такие как разрешение на чтение, создание, изменение или удаление определенных типов данных в приложении.

Многие службы управления идентификацией и доступом поддерживают стандарт JWT. Но их можно использовать и автономно, если вы решите реализовать свою службу авторизации. В этой главе мы будем применять службу Auth0.

Сначала создадим веб-токен JWT для представления некоторых утверждений о пользователе, а затем изменим предыдущее простейшее решение авторизации, добавим проверку токена и гарантируем доступ к GraphQL API только авторизованным пользователям. Для этого воспользуемся онлайн-отладчиком JWT по адресу **https://jwt.io**.

Нам понадобится случайная строка, которая будет применяться в качестве ключа подписи. Позже мы используем ее на сервере GraphQL для проверки входящих JWT:

```
Dpwm9XXKqk809WXjCsOmRSZQ5i5fXw8N
```

Введите это значение в разделе **VERIFY SIGNATURE** отладчика JWT. Затем добавьте несколько утверждений в полезную нагрузку токена (рис. 7.2):

```
{
  "sub": "1234567890",
  "name": "William Lyon",
  "email": "will@grandstack.io",
  "iat": 1638331785
}
```

Рис. 7.2. Создание подписанного веб-токена JWT с помощью jwt.io

Создав веб-токен JWT, вернемся к серверу GraphQL и добавим проверку токена. Сначала установим пакет `jsonwebtoken`:

```
npm install jsonwebtoken
```

Далее обновим логику функции разрешения и декодируем веб-токен JWT, используя ключ клиента, как показано в листинге 7.2.

Листинг 7.2. api/naive.js: проверка токена на сервере GraphQL

```
const { ApolloServer } = require("apollo-server");
const jwt = require("jsonwebtoken");

const peopleArray = [
  {
    name: "Bob",
  },
  {
    name: "Lindsey",
  },
];

const typeDefs = /* GraphQL */ `
```

```
  type Query {
    people: [Person]
  }

  type Person {
    name: String
  }
`;

const resolvers = {
  Query: {
    people: (obj, args, context, info) => {
      if (context.user) {
        return peopleArray;
      } else {
        throw new Error("You are not authorized");
      }
    },
  },
};

const server = new ApolloServer({
  resolvers,
  typeDefs,
  context: ({ req }) => {
    let decoded;
    if (req && req.headers && req.headers.authorization) {
      try {
        decoded = jwt.verify(
          req.headers.authorization.slice(7),
          "Dpwm9XXKqk809WXjCsOmRSZQ5i5fXw8N"
        );
      } catch (e) {
        // токен недействительный
        console.log(e);
      }
    }
    return {
      user: decoded,
    };
  },
});

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

Проверка токена с использованием ключа клиента

Если проверка, что токен подписан соответствующим ключом, прошла успешно, то обработка запроса продолжается и функция разрешения извлекает данные. Если токен недействителен, то клиенту возвращается ошибка и данные не извлекаются (рис. 7.3).

Рис. 7.3. Процесс проверки веб-токена JWT

В этом примере используется алгоритм шифрования HS256, соответственно, клиент и сервер используют один и тот же ключ. Позже, когда мы задействуем службу авторизации Auth0, мы используем более безопасный алгоритм RS256, в котором применяется пара ключей – открытый и закрытый.

Перезапустите сервер GraphQL, чтобы изменения вступили в силу, откройте Apollo Studio и добавьте токен JWT в заголовок авторизации. Если попытаться выполнить запрос без токена или с использованием недопустимого токена, сервер вернет ошибку: `You are not authorized` (Вы не авторизованы). Таким образом, сервер GraphQL будет выполнять только действительные запросы, содержащие JWT, подписанный с использованием закрытого ключа, соответствующего открытому ключу (рис. 7.4).

Рис. 7.4. Добавление JWT в заголовок Authorization в Apollo Studio

Выше упоминалось два недостатка нашего простейшего решения. Во-первых, мы не могли проверить действительность токена авторизации. Мы исправили этот недостаток, используя и проверив JWT, поэтому теперь исправим недостаток смешивания логики авторизации с логикой извлечения данных. Для этого используем директиву, с помощью которой определим правила авторизации в схеме GraphQL, и обеспечим их применение библиотекой Neo4j GraphQL.

7.3. Директива схемы @auth

Оставим позади простейшее решение авторизации на сервере GraphQL, вернемся к приложению обзора компаний и посмотрим, как добавить правила авторизации в схему GraphQL. Выше, на примере директивы `@cypher`, мы видели, как можно указать, что при разрешении запроса GraphQL должна применяться некоторая пользовательская логика на стороне сервера.

Библиотека Neo4j GraphQL включает директиву `@auth`, с помощью которой можно определять правила авторизации для защиты полей или типов в схеме GraphQL. Но прежде чем использовать директиву `@auth`, нужно определить метод и ключ для проверки JWT. Давайте объявим переменную окружения и присвоим ей значение нашего ключа:

```
export JWT_SECRET=Dpwm9XXKqk809WXjCsOmRSZQ5i5fXw8N
```

Затем обновим конфигурацию библиотеки Neo4j GraphQL, указав, что этот токен следует использовать для проверки авторизации (листинг 7.3). Для этого прочитаем только что созданную переменную окружения `JWT_SECRET` и передадим ее в объект `plugins` вместе с определениями типов и функций разрешения. Мы также должны установить пакет `graphql-plugin-auth` с плагинами авторизации для библиотеки Neo4j GraphQL:

```
npm i @neo4j/graphql-plugin-auth
```

Листинг 7.3. api/index.js: настройка авторизации в конфигурации библиотеки Neo4j GraphQL

```
const {
  Neo4jGraphQLAuthJWTPlugin,
} = require("@neo4j/graphql-plugin-auth");

const neoSchema = new Neo4jGraphQL({
  typeDefs,
  resolvers,
  driver,
  plugins: {
    auth: new Neo4jGraphQLAuthJWTPlugin({
      secret: process.env.JWT_SECRET,
    }),
  },
});
```

Проверка веб-токена JWT с помощью ключа

Еще можно настроить декодирование и проверку JWT, используя URL набора веб-ключей JSON Web Key Set (JWKS), что более безопасно, чем использование общего ключа. Мы применим этот метод для проверки JWT при использовании Auth0, а пока ограничимся решением с общим ключом (листинг 7.4). Нам также необходимо просмотреть объект HTTP-запроса, включающий заголовок авторизации и токен авторизации пользователя.

Листинг 7.4. api/index.js: просмотр объекта запроса в поисках токена авторизации

```
neoSchema.getSchema().then((schema) => {
  const server = new ApolloServer({
    schema,
    context: ({ req }) => ({ req }),
  });
  server.listen().then(({ url }) => {
    console.log(`GraphQL server ready at ${url}`);
  });
});
```

Передать объект HTTP-запроса в объект контекста, чтобы функции разрешения, сгенерированные библиотекой Neo4j GraphQL, могли декодировать JWT

7.3.1. Правила и операции

Используя директиву `@auth`, необходимо учитывать два аспекта: правила и операции. Оба передаются директиве в виде аргументов. Есть несколько типов правил авторизации, которые применяются в зависимости от того, как именно предполагается защищать поля и типы. Иногда желательно, чтобы определенные поля были доступны только пользователям, авторизовавшимся в системе. Иногда требуется, чтобы определенные типы могли править только администраторы. Или, например, чтобы только авторы отзывов могли обновлять их. Все это – правила авторизации, которые можно указать с помощью директивы `@auth`. В директиве `@auth` можно задать следующие типы правил:

- `isAuthenticated` – самое простое правило из поддерживаемых. Запрос GraphQL, обращающийся к защищенному типу или полю, должен иметь действительный JWT;
- `roles` – это правило задает одну или несколько ролей, которые должны быть определены в полезной нагрузке JWT;
- `allow` – это правило сравнивает значения полезной нагрузки в JWT со значениями в базе данных, и запрос считается действительным, только если эти значения равны;
- `bind` – это правило используется для сравнения значений в полезной нагрузке JWT и в операции мутации перед фиксацией в базе данных;
- `where` – это правило похоже на `allow` тем, что использует значение из полезной нагрузки JWT; однако вместо проверки на равенство добавляет предикат для фильтрации данных, соответствующих правилу.

При добавлении правил с помощью директивы `@auth` можно дополнительно указать одну или несколько операций, к которым должны применяться эти пра-

вила. Если операции не указаны, то правило будет применяться ко всем операциям. Можно использовать следующие операции:

- CREATE;
- READ;
- UPDATE;
- DELETE;
- CONNECT;
- DISCONNECT.

Давайте посмотрим, как действует директива @auth, чтобы понять, как следует использовать правила и операции в нашем приложении.

7.3.2. Правило авторизации isAuthenticated

Правило isAuthenticated может применяться как к типам, так и к полям GraphQL. Оно требует наличия действительного JWT для доступа к указанному типу или полю. Действительность JWT определяется возможностью проверить его с использованием ключа. Такая возможность указывает, что токен подписан закрытым ключом и создан авторизованным лицом. Логика isAuthenticated используется для блокировки областей приложения, требующих от пользователя аутентификации в приложении, но не требующих каких-либо конкретных разрешений – пользователю достаточно просто аутентифицироваться.

Допустим, мы решили в нашем приложении дать любому пользователю возможность искать компании, но показывать поле averageStars только аутентифицированным пользователям, побуждая пользователей регистрироваться в приложении. Давайте обновим наши определения типов GraphQL, чтобы включить это правило (листинг 7.5).

Листинг 7.5. api/index.js: изменения в типе Business

```
type Business {
  businessId: ID!
  waitTime: Int! @computed
  averageStars: Float
    @auth(rules: [{ isAuthenticated: true }])
    @cypher(
      statement: "MATCH (this)<-[:REVIEWS]-(r:Review) RETURN avg(r.stars)"
    )

  recommended(first: Int = 1): [Business]
    @cypher(
      statement: """
      MATCH (this)<-[:REVIEWS]-(:Review)<-[:WROTE]-(u:User)
      MATCH (u)-[:WROTE]->(:Review)-[:REVIEWS]->(rec:Business)
      WITH rec, COUNT(*) AS score
      RETURN rec ORDER BY score DESC LIMIT $first
      """
    )
```

Директива схемы @auth для защиты поля mediumStars с помощью правила isAuthenticated

```
  name: String!
  city: String!
  state: String!
  address: String!
  location: Point!
  reviews: [Review!]! @relationship(type: "REVIEWS", direction: IN)
  categories: [Category!]!
    @relationship(type: "IN_CATEGORY", direction: OUT)
}
```

Мы защитили доступ к полю `averageStars`, а это значит, что нужно включить допустимый JWT в заголовок любого запроса GraphQL с этим полем, как показано в листинге 7.6.

Листинг 7.6. Пример извлечения защищенного поля averageStars в запросе GraphQL

```
{
  businesses {
    name
    categories {
      name
    }
    averageStars
  }
}

"errors": [
  {
    "message": "Unauthenticated",
```

Если поле `averageStars` отсутствует в списке выбираемых полей, то запрос вернет ожидаемый результат. Поэкспериментируйте с отправкой недопустимого токена и запросов с полем `averageStars` и без него. Здесь мы включили действительный токен в заголовок авторизации запроса, позволяющий просматривать поле `averageStars`:

```
{
"Authorization":"Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxM
jM0NTY3ODkwIiwibmFtZSI6IldpbGxpYW0gTHlvbiIsImVtYWlsIjoid2lsbEBncmFuZHN0YWNr
LmlvIiwiaWF0IjoxNTE2MjM5MDIyfQ.Y37P8OF_qMamIcZldi89Nm0YQdF4v91iHQWrNu0jtBk"
}
```

7.3.3. Правило авторизации roles

Правило `roles` позволяет определять требования к типам разрешений, необходимых для одной или нескольких операций. Чтобы получить доступ к полю или типу, защищенному правилом `roles`, необходимо иметь не просто действительный подписанный токен, но токен должен включать одну из ролей, указанных в утверждении `roles`. Рассмотрим пример в листинге 7.7.

Листинг 7.7. api/index.js: защита типа User требованием обладать определенной ролью

```
extend type User @auth(rules: [{roles: ["admin"]}])
```

Здесь использовано ключевое слово `extend`, чтобы добавить дополнительные директивы или поля к уже определенному типу. Этот синтаксис эквивалентен включению директивы в определение типа, но использование `extend` позволяет разнести определения типов по нескольким файлам, если вдруг возникнет такое желание (рис. 7.5).

Рис. 7.5. Процесс авторизации при использовании директивы @auth

Теперь любая операция GraphQL, обращающаяся к типу `User`, должна выполняться пользователем с ролью `admin`, включая операции, выполняющие обход пользователей, как показано в листинге 7.8.

Листинг 7.8. Доступ к информации о пользователях в запросах GraphQL

```
query {
  businesses(where: {name: "Neo4j"}) {
    name
```

```
    categories {
      name
    }
    address
    reviews {
      text
      stars
      date
      user {
        name
      }
    }
  }
}
```

При попытке выполнить предыдущий запрос мы получим следующее сообщение об ошибке из-за того, что наш токен не содержит роли `admin`:

```
"errors": [
  {
    "message": "Forbidden"
  }
]
```

Добавим роли в утверждения в токене. Для этого в онлайн-конструкторе JWT (**https://jwt.io**) добавьте массив `roles` в список утверждений:

```
{
  "sub": "1234567890",
  "name": "William Lyon",
  "email": "will@grandstack.io",
  "iat": 1516239022,
  "roles": ["admin"]
}
```

Теперь, вставив этот обновленный токен в заголовок авторизации в Apollo Studio и снова запустив запрос GraphQL, мы получим доступ к информации о пользователе.

Напомню, что если не указываются конкретные операции (например, создание, чтение и обновление), то правила авторизации применяются ко всем операциям, включающим данный тип или поле. Чтобы ограничить применение правила только к определенным операциям, нужно явно указать их в определении правила в директиве `@auth`.

Два рассматривавшихся нами правила `@auth` (`isAuthenticated` и `roles`) использовали только значения из полезной нагрузки JWT (или, в случае `isAuthenticated`, просто проверяли наличие действительного токена). Следующие три правила, которые мы рассмотрим, будут использовать значения из базы данных, чтобы обеспечить соблюдение правил авторизации.

7.3.4 Правило авторизации allow

Выше мы определили правило, защищающее тип `User`, требуя, чтобы аутентифицированный пользователь имел роль `admin`. Теперь добавим дополнительное правило, позволяющее пользователям читать информацию о самих себе.

Листинг 7.9. api/index.js: разрешение пользователям доступа к информации о самих себе

```
extend type User
  @auth(
    rules: [
      { operations: [READ], allow: { userId: "$jwt.sub" } }
      { roles: ["admin"] }
    ]
  )
```

Обратите внимание, что здесь новое правило `allow` объединено с существующим правилом `roles`. Аргумент `rules` принимает массив правил и объединяет их логической операцией ИЛИ. Чтобы получить доступ к типу `User`, утверждения в JWT должны соответствовать хотя бы одному из правил авторизации, определенных в массиве `rules`. В этом случае аутентифицированный пользователь должен быть либо администратором, либо его идентификатор должен совпадать с `userId` запрашиваемого узла `User`. Чтобы протестировать новое правило, создадим новый JWT для пользователя Jenny со следующей полезной нагрузкой:

```
{
  "sub": "u3",
  "name": "Jenny",
  "email": "jenny@grandstack.io",
  "iat": 1516239022,
  "roles": [
    "user"
  ]
}
```

Создать его можно с помощью веб-интерфейса **jwt.io**; просто используйте тот же ключ JWT для подписания токена:


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1MyIsIm5hbWUiOiJKZW5ueSIsIm
VtYWlsIjoiamVubnlAZ3JhbmRzdGFjay5pbyIsImlhdCI6MTUxNjIzOTAyMiwicm9sZXMiOlsid
XNlciJdfQ.ctal5qgshR4-hqchxsYxxHVGPsE0JNxydGy3Pga27nA
```


Теперь, применяя этот JWT, выполним запрос GraphQL от имени пользователя Jenny, как показано в листинге 7.10.

Листинг 7.10. Запрос информации о единственном пользователе

```
query {
  users(where: { name: "Jenny" }) {
```

```
    name
    userId
  }
}
```

Поскольку вложенное утверждение в JWT соответствует идентификатору `userId` пользователя, от имени которого выполняется запрос, в ответ сервер вернет хранящуюся в нем информацию:

```
{
  "data": {
    "users": [
      {
        "name": "Jenny",
        "userId": "u3"
      }
    ]
  }
}
```

В этом случае запрос GraphQL выполняет фильтрацию по имени пользователя, используя аргумент `where`, и возвращает только те данные, к которым у нас есть доступ. Но что произойдет, если запросим данные, к которым у нас нет доступа? Для примера попробуем запросить информацию обо всех пользователях (листинг 7.11).

Листинг 7.11. Запрос информации обо всех пользователях

```
query {
  users {
    name
    userId
  }
}
```

Пользователь, от имени которого выполняется запрос, не является администратором, и запрашиваются объекты `User`, `userId` в которых не соответствует утверждению в JWT, поэтому мы получим ошибку `Forbidden`.

Давайте посмотрим, как избежать таких ошибок, автоматически фильтруя результаты запроса, чтобы вернуть только те данные, к которым имеет доступ аутентифицированный пользователь. Для этого используем правило авторизации `where`. Это решение позволит клиенту не беспокоиться о добавлении фильтра, чтобы ненароком не запросить данные, к которым у пользователя нет доступа.

7.3.5. Правило авторизации where

В предыдущем разделе мы использовали правило `allow`, чтобы пользователи могли извлекать только свои собственные данные. Однако это довольно проблематичный подход, потому что требует от клиента позаботиться о добавлении соответствующих фильтров, чтобы не запросить данные, к которым у него нет

доступа. Вместо этого воспользуемся правилом `where` (листинг 7.12), избавляющим от лишних хлопот.

Листинг 7.12. api/index.js: использование правила авторизации `where`

```
extend type User
  @auth(
    rules: [
      { operations: [READ], where: { userId: "$jwt.sub" } }
      { operations: [CREATE, UPDATE, DELETE], roles: ["admin"] }
    ]
  )
```

Нам все так же нужно гарантировать, чтобы только пользователи с правами администратора могли создавать, обновлять или удалять пользователей, поэтому добавим эти операции в правило `roles` (листинг 7.13). Теперь при выполнении запроса на чтение типа `User` в сгенерированный запрос к базе данных будет добавляться предикат, фильтрующий данные и оставляющий только ту информацию, которая доступна текущему аутентифицированному пользователю, путем сопоставления вложенного утверждения JWT со значением свойства `userId` узла `User` в базе данных.

Листинг 7.13. Запрос GraphQL на получение информации о пользователе

```
query {
  users {
    name
    userId
  }
}
{
  "data": {
    "users": [
      {
        "name": "Jenny",
        "userId": "u3"
      }
    ]
  }
}
```

Взглянув на сгенерированный запрос Cypher, отправляемый в базу данных, можно заметить добавленный предикат, сопоставляющий значение свойства `userId` узла `User` в базе данных со значением `sub` в JWT, как показано в листинге 7.14.

Листинг 7.14. Сгенерированный запрос Cypher

```
MATCH (this:User)
WHERE this.userId IS NOT NULL AND this.userId = "u3"
RETURN this { .name, .userId } as this
```

7.3.6. Правило авторизации bind

Правило `bind` используется для принудительного применения правил авторизации при создании или обновлении данных, а также может использоваться в отношениях. В листинге 7.15 показано, как задействовать правило `bind`, чтобы гарантировать, что при создании или обновлении отзывов они автоматически связывались с текущим аутентифицированным пользователем. Мы должны предотвратить возможность создания отзывов от имени другого пользователя!

Листинг 7.15. api/index.js: использование правила авторизации bind

```
extend type Review
  @auth(
    rules: [
      {
        operations: [CREATE, UPDATE]
        bind: { user: { userId: "$jwt.sub" } }
      }
    ]
)
```

Теперь напишем запрос-мутацию GraphQL, создающий новый отзыв (листинг 7.16).

Листинг 7.16. Создание нового отзыва

```
mutation {
  createReviews(
    input: {
      business: { connect: { where: { node: { businessId: "b10" } } } }
      date: "2022-01-22"
      stars: 5.0
      text: "Love the Philtered Soul!"
      user: { connect: { where: { node: { userId: "u3" } } } }
    }
  ) {
    reviews {
      business {
        name
      }
      text
      stars
    }
  }
}
```

Этот запрос успешно выполняется и добавляет в базу данных узел `Review` и соответствующие отношения:

```
{
  "data": {
    "createReviews": {
      "reviews": [
        {
          "business": {
            "name": "Philz Coffee"
          },
          "text": "Love the Philtered Soul!",
          "stars": 5
        }
      ]
    }
  }
}
```

Однако если вместо текущего аутентифицированного пользователя (в данном случае пользователя с `userId u3`) операция мутации попытается связать отзыв с пользователем с `userId u1` или вообще ни с одним пользователем, то она завершится ошибкой и сообщением `Forbidden`.

Дополнительные примеры добавления сложных правил авторизации в GraphQL API вы найдете в документации с описанием директивы `@auth`: **https://neo4j.com/docs/graphql-manual/3.0/auth/**.

До сих пор для создания JWT мы использовали веб-сайт JWT Builder; это удобно на этапе разработки и тестирования, но для производства нужно что-то большее.

7.4. Auth0: JWT как услуга

Auth0 – это служба аутентификации и авторизации, позволяющая аутентифицировать пользователей несколькими способами, например через вход в социальные сети, по адресу электронной почты и паролю и т. д. Она также включает функции ведения базы данных пользователей, и с ее помощью можно определять правила и разрешения для наших пользователей. Я предпочитаю рассматривать Auth0 как поставщика JWT как услуги. Несмотря на то что Auth0 имеет множество функций, мне чаще интересно только получить токен аутентификации пользователя (в виде JWT) и использовать его для авторизации пользователя в моих API и приложениях.

Auth0 также хорошо подходит для обучения и разработки, потому что предлагает бесплатный тариф без указания кредитной карты при регистрации. В этом разделе мы настроим Auth0 для защиты нашего API, а затем используем Auth0 React SDK, чтобы добавить поддержку Auth0 в наше приложение. Вы можете бесплатно создать учетную запись Auth0 на странице **https://auth0.com**.

7.4.1. *Настройка Auth0*

После входа в Auth0 нужно создать проект API и приложения (рис. 7.6). Сначала создайте API и дайте ему имя.

Рис. 7.6. Создание API в Auth0

В нашем приложении не будет использоваться управление доступом на основе ролей (Role-Based Access Control, RBAC), но в общем случае есть возможность дополнительно включить эту функцию для поддержки нашего API (рис. 7.7). Это позволит добавлять подробные разрешения для JWT, сгенерированные Auth0, которые могут использоваться правилом `roles` директивы `@auth` для управления доступом на основе ролей.

Если включить поддержку RBAC, то также нужно определить все возможные разрешения в нашем API. Я добавил необходимые разрешения для создания, чтения, обновления и удаления компаний в нашем API (рис. 7.8).

Дополнительную информацию об использовании правила авторизации `roles` совместно с поддержкой RBAC можно найти в документации библиотеки Neo4j GraphQL: **http://mng.bz/5Q5z**.

Теперь нужно создать приложение в панели инструментов Auth0. Выберите **Create Application** (Создать приложение) и укажите имя нашего приложения – я использовал имя *Business Reviews*. Вам также будет предложено выбрать тип приложения. Поскольку мы создаем приложение React, выберите **Single Page Web Application** (Одностраничное веб-приложение) и щелкните на кнопке **Create** (Создать).

Мы оставим большинство настроек по умолчанию, но изменим записи в разделах **Allowed Callback URLs** (Разрешенные URL для обратных вызовов) и **Allowed Logout URLs** (Разрешенные URL выхода). Добавьте `http://localhost:3000` в каждое из этих текстовых полей на вкладке **Settings** (Настройки), а затем выберите **Save Changes** (Сохранить изменения).

RBAC Settings

Enable RBAC — ENABLED

If this setting is enabled, RBAC authorization policies will be enforced for this API. Role and permission assignments will be evaluated during the login transaction.

Add Permissions in the Access Token — ENABLED

If this setting is enabled, the Permissions claim will be added to the access token. Only available if RBAC is enabled for this API.

Access Settings

Allow Skipping User Consent — ENABLED

If this setting is enabled, this API will skip user consent for applications flagged as First Party.

Allow Offline Access — DISABLED

If this setting is enabled, Auth0 will allow applications to ask for Refresh Tokens for this API.

SAVE

Рис. 7.7. Включение поддержки RBAC для нашего API в Auth0

Business Reviews API

CUSTOM API Identifier https://reviews.grandstack.io

Quick Start | Settings | Permissions | Machine to Machine Applications | Test

Add a Permission (Scope)

Define the permissions (scopes) that this API uses.

Permission (Scope) * read:appointments

Description * Read your appointments

+ ADD

List of Permissions (Scopes)

These are all the permissions (scopes) that this API uses.

Permission	Description
read:business	Read business data
create:business	Create business nodes
update:business	Update existing businesses
delete:business	Delete business nodes

Рис. 7.8. Добавление разрешений для API в Auth0

Далее нужно обновить конфигурацию в нашем GraphQL API и указать метод проверки JWT, сгенерированный службой Auth0 (листинг 7.17). До сих пор для проверки JWT мы использовали простой ключ, хранящийся в переменной окружения (`JWT_SECRET`). Это прием годится для локальной разработки и тестирования, но теперь, когда мы используем Auth0 и готовимся развернуть наше приложение в интернете, нам нужен более безопасный метод.

Перейдите на вкладку **Advanced Settings** (Дополнительные настройки), а затем в раздел к **Endpoints** (Конечные точки). Найдите URL JWKS и скопируйте это значение. Затем в коде GraphQL API измените метод, используемый для проверки JWT, на `jwksEndpoint`, подставив URL вашего приложения Auth0. Это позволит нашему GraphQL API получить открытый ключ от Auth0 для проверки токена, что является гораздо безопасным методом, чем использование общего ключа.

Листинг 7.17. api/index.js: использование конечной точки JSON Web Key Set (JWKS) службы Auth0

```
const {
  Neo4jGraphQLAuthJWKSPlugin,
} = require("@neo4j/graphql-plugin-auth");

...

const neoSchema = new Neo4jGraphQL({
  typeDefs,
  resolvers,
  driver,
  plugins: {
    auth: new Neo4jGraphQLAuthJWKSPlugin({
      jwksEndpoint: "https://grandstack.auth0.com/.well-known/jwks.json",
    }),
  },
});
```

Теперь следует использовать класс `Neo4jGraphQLAuthJWKSPlugin`

В дополнительных настройках Auth0 обязательно укажите свою конечную точку

Теперь мы готовы начать интеграцию Auth0 в наше приложение React.

7.4.2. Auth0 React

Сначала установим Auth0 React SDK. Этот пакет включает все необходимое для интеграции внедрения поддержки Auth0 в любые приложения React.

Установим библиотеку `auth0-react` с помощью `npm`, но не забудьте предварительно перейти в каталог `web-react`:

```
npm install @auth0/auth0-react
```

Теперь добавим начальные настройки Auth0 в приложение React (листинг 7.18).

Листинг 7.18. web-react/src/index.js: добавление компонента Auth0 Provider

```
import React from "react";
import ReactDOM from "react-dom";
```

```
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
import {
  ApolloClient,
  InMemoryCache,
  ApolloProvider,
  makeVar,
} from "@apollo/client";
import { Auth0Provider } from "@auth0/auth0-react";

export const starredVar = makeVar([]);

const client = new ApolloClient({
  uri: "http://localhost:4000",
  cache: new InMemoryCache({
    typePolicies: {
      Business: {
        fields: {
          isStarred: {
            read(_, { readField }) {
              return starredVar().includes(readField("businessId"));
            },
          },
        },
      },
    },
  }),
});

ReactDOM.render(
  <React.StrictMode>
    <Auth0Provider
      domain="grandstack.auth0.com"
      clientId="4xw3K3cjvw0hyT4Mjp4RuOVSxvVYcOFF"
      redirectUri={window.location.origin}
      audience="https://reviews.grandstack.io"
    >
      <ApolloProvider client={client}>
        <App />
      </ApolloProvider>
    </Auth0Provider>
  </React.StrictMode>,
  document.getElementById("root")
);

// Чтобы оценить работу приложения, вызовите функцию журналирования
```

Импорт компонента Auth0 Provider

Заключить компонент приложения в компонент `Auth0Provider`

```
// результатов (например: reportWebVitals(console.log))
// или передайте их конечной точке analytics. Дополнительную информацию
// ищите по адресу: https://bit.ly/CRA-vitals
reportWebVitals();
```

Мы добавили компонент `Auth0Provider`, внедрив его в иерархию компонентов, и заключили в него наши компоненты `ApolloProvider` и `App`. Настроили домен, идентификатор клиента и информацию об аудитории для только что созданных клиента Auth0, приложения и API. Эту информацию можно найти на панели инструментов Auth0.

В листинге 7.19 добавляются кнопки входа и выхода из приложения с использованием Auth0. Щелчок на кнопке входа запустит процесс аутентификации Auth0.

Листинг 7.19. web-react/src/App.js: добавление кнопок входа и выхода

```
import React, { useState } from "react";
import BusinessResults from "./BusinessResults";
import { gql, useQuery } from "@apollo/client";
import { useAuth0 } from "@auth0/auth0-react";

const GET_BUSINESSES_QUERY = gql`
  query BusinessesByCategory($selectedCategory: String!) {
    businesses(
      where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
    ) {
      businessId
      name
      address
      categories {
        name
      }
      isStarred @client
    }
  }
`;

function App() {
  const [selectedCategory, setSelectedCategory] = useState("");
  const { loginWithRedirect, logout, isAuthenticated } = useAuth0();

  const { loading, error, data, refetch } = useQuery(
    GET_BUSINESSES_QUERY,
    {
      variables: { selectedCategory },
    }
```

Импортировать обработчик `useAuth0` **из библиотеки React**

Функции поддержки аутентификации и доступа к пользовательским данным

```
  );

  if (error) return <p>Error</p>;
  if (loading) return <p>Loading...</p>;

  return (
    <div>
      {!isAuthenticated && (
        <button onClick={() => loginWithRedirect()}>Log In</button>
      )}
      {isAuthenticated && (
        <button onClick={() => logout()}>Log Out</button>
      )}
      <h1>Business Search</h1>
      <form>
        <label>
          Select Business Category:
          <select
            value={selectedCategory}
            onChange={(event) => setSelectedCategory(event.target.value)}
          >
            <option value="">All</option>
            <option value="Library">Library</option>
            <option value="Restaurant">Restaurant</option>
            <option value="Car Wash">Car Wash</option>
          </select>
        </label>
        <input type="button" value="Refetch" onClick={() => refetch()} />
      </form>

      <BusinessResults businesses={data.businesses} />
    </div>
  );
}

export default App;
```

Добавление кнопок входа и выхода

Пакет Auth0 React включает обработчик `useAuth0`, дающий доступ к функциям, способным запускать процесс аутентификации, определять, аутентифицирован ли пользователь в данный момент, и получать информацию о пользователе. Теперь у нас есть кнопка для входа и для выхода, которые появляются перед входом и после него соответственно.

После щелчка на кнопке **Log In** (Войти) предлагается несколько вариантов входа, в том числе через GitHub, Google, Twitter или с адресом электронной почты и паролем (рис. 7.9). Одно из преимуществ использования внешней службы аутентификации – не нужно беспокоиться об особенностях процесса аутентификации, потому что обо всем этом позаботится Auth0.

Рис. 7.9. Варианты входа, предлагаемые службой Auth0

Обратите внимание, как используется переменная `isAuthenticated`, предоставляемая обработчиком `useAuth0`. Сразу после входа также можно получить информацию о пользователе. Давайте добавим компонент профиля, отображающий имя пользователя и его аватар после входа в систему. Создадим новый файл Profile.js в каталоге web-react/src, как показано в листинге 7.20.

Листинг 7.20. web-react/src/Profile.js: добавление компонента профиля пользователя

```
import { useAuth0 } from "@auth0/auth0-react";

const Profile = () => {
  const { user, isAuthenticated } = useAuth0();
  return (
    isAuthenticated && (
      <div style={{ padding: "10px" }}>
        <img
          src={user.picture}
          alt="User avatar"
          style={{ width: "40px" }}
        />
        <strong>{user.name}</strong>
      </div>
    )
  );
};

export default Profile;
```

Теперь включим компонент профиля в основной компонент App, чтобы отображать информацию из профиля, после того как пользователь выполнит вход (листинг 7.21).

Листинг 7.21. web-react/src/App.js: добавление компонента профиля

```
import Profile from "./Profile";

...

      {!isAuthenticated && (
        <button onClick={() => loginWithRedirect()}>Log In</button>
      )}
      {isAuthenticated && <button onClick={() => logout()}>Log Out</button>}
      <Profile />
      <h1>Business Search</h1>

...
```

Добавление компонента Profile

Итак, мы добавили отображение информации о пользователе после входа в приложение, как показано на рис. 7.10, но как выполнять аутентифицированные запросы к GraphQL API? Используя Apollo Studio, мы видели, что в заголовке запроса GraphQL нужно передать токен авторизации.

Рис. 7.10. Вид приложения React, после того как пользователь выполнил вход

Получить токен можно вызовом функции getAccessTokenSilently из библиотеки auth0-react и затем прикрепить его к экземпляру Apollo Client, как показано в листинге 7.22.

Листинг 7.22. web-react/src/index.js: добавление токена доступа в запрос GraphQL

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import App from "./App";
import reportWebVitals from "./reportWebVitals";
import {
  ApolloClient,
  InMemoryCache,
  ApolloProvider,
  makeVar,
  createHttpLink,
} from "@apollo/client";
import { setContext } from "@apollo/client/link/context";
import { Auth0Provider, useAuth0 } from "@auth0/auth0-react";

export const starredVar = makeVar([]);

const AppWithApollo = () => {
```

Создание компонента-обертки, отвечающего за добавление токена авторизации

```
  const { getAccessTokenSilently, isAuthenticated } = useAuth0();

  const httpLink = createHttpLink({
    uri: "http://localhost:4000",
  });

  const authLink = setContext(async (_, { headers }) => {
```

Вызов функции `setContext` из Apollo Client и добавление JWT в запрос GraphQL

```
    // Получить токен можно только после аутентификации пользователя
    const accessToken = isAuthenticated
      ? await getAccessTokenSilently()
      : undefined;
    if (accessToken) {
      return {
        headers: {
          ...headers,
          authorization: accessToken ? `Bearer ${accessToken}` : "",
        },
      };
    } else {
      return {
        headers: {
          ...headers,
          // Здесь можно установить дополнительные заголовки или,
          // при необходимости, заголовок авторизации "по умолчанию"
        },
      };
```

```
    }
  });

  const client = new ApolloClient({
    link: authLink.concat(httpLink),
    cache: new InMemoryCache({
      typePolicies: {
        Business: {
          fields: {
            isStarred: {
              read(_, { readField }) {
                return starredVar().includes(readField("businessId"));
              },
            },
          },
        },
      },
    }),
  });

  return (
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>
  );
};

ReactDOM.render(
  <React.StrictMode>
    <Auth0Provider
      domain="grandstack.auth0.com"
      clientId="4xw3K3cjvw0hyT4Mjp4RuOVSxvVYcOFF"
      redirectUri={window.location.origin}
      audience="https://reviews.grandstack.io"
    >
      <AppWithApollo />
    </Auth0Provider>
  </React.StrictMode>,
  document.getElementById("root")

reportWebVitals();
```

Вставка компонента `AppWithApollo` **в иерархию компонентов React**

Теперь в каждый запрос к GraphQL API будет добавляться заголовок с токеном авторизации, если пользователь аутентифицирован. Чтобы убедиться в этом, откройте инструменты разработчика в браузере и посмотрите, как выглядит сетевой запрос к GraphQL (рис. 7.11).

Рис. 7.11. Вид заголовка авторизации, добавленного в запрос GraphQL

Этот токен можно скопировать и расшифровать его полезную нагрузку с помощью **jwt.io**. Вот как выглядит мой декодированный токен:

```
{
  "iss": "https://grandstack.auth0.com/",
  "sub": "github|1222454",
  "aud": [
    "https://reviews.grandstack.io",
    "https://grandstack.auth0.com/userinfo"
  ],
  "iat": 1599684745,
  "exp": 1599771145,
  "azp": "4xw3K3cjvw0hyT4Mjp4RuOVSxvVYcOFF",
  "scope": "openid profile email"
}
```

Поведение нашего приложения не изменилось, потому что оно пока не запрашивает никаких защищенных полей. Давайте добавим поле `averageStars`, защи-

щенное правилом `isAuthenticated`, в запрос GraphQL, если он выполняется после входа пользователя (листинг 7.23).

Листинг 7.23. web-react/src/App.js: включение поля `averageStars` в запрос

```
function App() {
  const [selectedCategory, setSelectedCategory] = useState("");
  const { loginWithRedirect, logout, isAuthenticated } = useAuth0();

  const GET_BUSINESSES_QUERY = gql`
  query BusinessesByCategory($selectedCategory: String!) {
    businesses(
      where: { categories_SOME: { name_CONTAINS: $selectedCategory } }
    ) {
      businessId
      name
      address
      categories {
        name
      }
      ${isAuthenticated ? "averageStars" : ""}
        isStarred @client
      }
    }
  `;

  const { loading, error, data, refetch } = useQuery(
    GET_BUSINESSES_QUERY,
    {
      variables: { selectedCategory },
    }
  );

  if (error) return <p>Error</p>;
  if (loading) return <p>Loading...</p>;
```

Добавьте поле `averageStars`, если пользователь аутентификацирован

А теперь обновим компонент `BusinessResults` и добавим в него поле `averageStars`, когда запрос выполняется аутентифицированным пользователем (листинг 7.24).

Листинг 7.24. web-react/src/BusinessResults.js: вывод поля averageStars

```
import React from "react";
import { starredVar } from "./index";
import { useAuth0 } from "@auth0/auth0-react";

function BusinessResults(props) {
  const { businesses } = props;
  const starredItems = starredVar();
```

```
  const { isAuthenticated } = useAuth0();

  return (
    <div>
      <h2>Results</h2>
      <table>
        <thead>
          <tr>
            <th>Star</th>
            <th>Name</th>
            <th>Address</th>
            <th>Category</th>
            {isAuthenticated ? <th>Average Stars</th> : null}
          </tr>
        </thead>
        <tbody>
          {businesses.map((b) => (
            <tr key={b.businessId}>
              <td>
                <button
                  onClick={() =>
                    starredVar([...starredItems, b.businessId])
                  }
                >
                  Star
                </button>
              </td>
              <td style={b.isStarred ? { fontWeight: "bold" } : null}>
                {b.name}
              </td>
              <td>{b.address}</td>
              <td>
                {b.categories.reduce(
                  (acc, c, i) => acc + (i === 0 ? " " : ", ") + c.name,
                  ""
                )}
              </td>
              {isAuthenticated ? <td>{b.averageStars}</td> : null}
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}

export default BusinessResults;
```

Добавлять заголовок `Average Stars`, **только если пользователь аутентифицирован**

Показывать среднее поля `averageStars`, **только если пользователь аутентифицирован**

Теперь среднее число звезд у каждой компании будут видеть только аутентифицированные пользователи. Мы добавили в приложение аутентификацию, авторизацию и поддержку Auth0. Теперь, защитив приложение, перейдем к следующей главе и посмотрим, как развернуть его и базу данных.

7.5. Упражнения

1. Создайте новое поле с именем `qualityBusinesses` и снабдите его директивой `@cypher` с запросом, возвращающим компании, которые имеют не менее двух отзывов и четыре или более звезд. Защитите это поле правилом `roles` и директивой `@auth`, потребовав наличие роли `analyst`. Создайте JWT, включающий эту роль в утверждения, и используйте Apollo Studio для запроса этого нового поля.
2. В этой главе мы использовали мутацию GraphQL, чтобы создать новый отзыв о компании. Добавьте в приложение React форму, позволяющую текущему аутентифицированному пользователю создавать новые отзывы.

Итоги

- Правила авторизации можно выражать декларативно с помощью директивы `@auth`.
- JWT – это стандарт кодирования и передачи объектов JSON; он широко используется для представления токенов авторизации в веб-приложениях, таких как GraphQL API.
- Auth0 – это служба управления идентификацией пользователей и их разрешениями. Ее можно использовать для создания токенов JWT и аутентификации пользователей. В приложения React служба Auth0 интегрируется с помощью Auth0 React SDK.

Глава 8

Развертывание приложения GraphQL

В этой главе:

- развертывание приложения GraphQL так, чтобы оно было доступно из интернета;
- развертывание в бессерверных окружениях и использование облачных служб, таких как Netlify, AWS Lambda и Neo4j Aura;
- платформа для оценки различных вариантов развертывания, помогающая выяснить и согласовать неизбежные компромиссы.

Разрабатывая приложение, мы запускали его локально на своей машине. Теперь пришло время развернуть приложение, чтобы поделиться им со всем миром и дать пользователям возможность взаимодействовать с ним. Существует множество способов развертывания приложений, в том числе и в облачных окружениях, предлагающих улучшенные условия для разработчиков. Не существует единственного варианта развертывания, лучшего для любых приложений, потому что каждый вариант имеет свои достоинства и недостатки; в конечном счете разработчик должен решить, какие варианты лучше подходят для него и его приложения.

В этой главе мы исследуем возможности развертывания нашего приложения GraphQL с использованием услуг сторонних поставщиков, таких как Netlify, AWS Lambda и Neo4j Aura. Такой подход к применению управляемых служб, когда большая часть операций делегируется этим поставщикам, часто называют *бессерверным* (serverless). Мы оценим преимущества и недостатки этого подхода, воспользовавшись платформой, ориентированной на операции, масштабирование и удобство разработки. Под конец рассмотрим альтернативные варианты развертывания и обсудим неизбежные компромиссы.

8.1. Развертывание приложения GraphQL

Бессерверные вычисления – это парадигма, описывающая способ распределения вычислительных ресурсов и выполнения по требованию; это способ доставки

приложений без необходимости подготавливать и обслуживать серверы. Такие службы, как платформа AWS Lambda Function as a Service (FaaS), считаются бессерверными, но не потому, что серверы не участвуют в процессе обслуживания приложений, а потому, что разработчику не приходится думать о серверах. Вместо этого соответствующей абстракцией становится *функция*. Значение термина *бессерверный* расширилось, и в настоящее время под ним подразумеваются не только вычислительные окружения, такие как AWS Lambda и Google App Engine, но также базы данных и другие управляемые облачные службы.

Первая парадигма развертывания, которую мы рассмотрим, использует преимущества *управляемых служб*. Управляемая служба – это способ передачи ответственности за настройку системного программного обеспечения, инфраструктуры или сети поставщику облачных услуг. При использовании управляемых служб разработчики могут тратить меньше времени на обслуживание баз данных, поддержку масштабирования веб-серверов, установку обновлений безопасности и SSL-сертификатов и вместо этого сосредоточиться на прикладных аспектах своего программного обеспечения, дающих им конкурентные преимущества. Наш подход особенно привлекателен для разработчиков приложений полного цикла, не имеющих экспертных знаний и не готовых взять на себя ответственность за управление базами данных, серверами и SSL-сертификатами, настройку DNS и другие аспекты, необходимые веб-приложениям.

8.1.1. *Преимущества развертывания в бессерверном окружении*

Использование управляемых служб имеет свои преимущества по сравнению с альтернативными подходами. Здесь мы особо выделим такие преимущества, как продуктивность разработчиков, стоимость использования, масштабируемость, обслуживание и эксплуатация.

Продуктивность разработчика

Многие управляемые службы гордятся тем, что предлагают дополнительные удобства для разработчиков, позволяя абстрагироваться от многих ненужных сложностей или проблем, не связанных с основными целями разработки: созданием и доставкой приложений. Такие инструменты, как веб-консоли для настройки служб и интерфейсы командной строки (Command Line Interface, CLI), которые можно интегрировать в процессы разработки, способствуют повышению продуктивности разработчиков.

Стоимость использования

Оплата услуг в зависимости от особенностей их использования является основным принципом этой парадигмы. Если приложение используется очень мало, то разработчик будет нести небольшие расходы. Это позволяет создавать, развертывать и тестировать приложения с небольшими первоначальными затратами, потому что их размер не является фиксированным.

Масштабируемость

Службы должны масштабироваться с увеличением их потребления. Например, среда выполнения FaaS, такая как AWS Lambda, запускается в ответ на события,

такие как вызов конечной точки API. Вызовы функций не имеют состояния и выполняются одновременно, что обеспечивает улучшенную эластичность и способность к масштабированию.

Техническое обслуживание и эксплуатация

При использовании управляемых служб ответственность за работоспособность, безопасность и высокую производительность службы возлагается на поставщика. Это преимущество часто находит отклик у разработчиков, обычно отвечающих за множество компонентов в приложениях.

8.1.2. *Недостатки развертывания в бессерверном окружении*

Управляемые службы – это не панацея, способная решить все проблемы, и, конечно же, имеются некоторые недостатки, к которым можно отнести зависимость от определенного поставщика, оптимизацию производительности и оплату, в зависимости от характера использования (палка о двух концах!).

Зависимость от поставщика

Передача поставщику ответственности за обслуживание и обновление службы означает, что разработчик находится в полной власти поставщика в части обеспечения непрерывной работы службы по разумной цене. Иногда службы могут прекращать существование или объявляться устаревшими, а так как многие службы имеют определенные API, библиотеки или парадигмы, адаптация приложения к альтернативным решениям может оказаться слишком дорогостоящей для разработчика.

Оптимизация производительности

Многие службы работают в архитектурах со множеством арендаторов, и не всегда есть возможность гарантировать постоянство производительности, потому что ресурсы могут использоваться совместно с другим пользователями. Учитывая природу работы по запросу, могут возникнуть некоторые накладные расходы, обусловленные выделением дополнительных ресурсов в ответ на возросшее потребление.

Стоимость использования

Стоимость использования может быть как преимуществом, так и недостатком. Если структура затрат и модели использования не изучены, или наблюдаются периодические всплески потребления, то затраты могут неожиданно увеличиваться.

8.1.3. *Обзор подхода к развертыванию приложения GraphQL в бессерверном окружении*

Наш выбор подхода к развертыванию в бессерверном окружении основан на преимуществах трех управляемых служб (рис. 8.1):

1) *база данных Neo4j Aura как услуга* позволяет развернуть управляемую и масштабируемую графовую базу данных в облаке. Neo4j Aura избавляет от не-

обходимости думать об управлении экземпляром базы данных. Такие операции, как регулярное резервное копирование и обновление, выполняются службой;

2) *Netlify Build* собирает, развертывает и обновляет приложение React, а также обеспечивает его глобальное обслуживание через сеть доставки контента (Content Delivery Network, CDN). Платформа Netlify не только даст доступ к глобальной CDN, обеспечивающей быструю загрузку нашего сайта, независимо от физического местонахождения пользователей, но также предложит удобства разработки и интеграцию с системами управления версиями, такими как GitHub;
3) *AWS Lambda (через Netlify Functions)* позволяет развернуть GraphQL API в форме масштабируемой бессерверной функции. Использование AWS Lambda для развертывания GraphQL API избавляет от необходимости думать о размещении веб-серверов и управлении ими, а также о масштабировании серверов вверх и вниз с изменением количества запросов, поступающих в единицу времени.

Рис. 8.1. Развертывание приложения GraphQL с точки зрения пользователя

8.2. База данных Neo4j Aura как услуга

Neo4j Aura – это управляемая облачная служба Neo4j кластеров баз данных. Neo4j Aura предлагает масштабируемые высокодоступные кластеры Neo4j и освобождает разработчиков от бремени эксплуатации и обслуживания. Разработчики могут создавать кластеры Neo4j одним нажатием кнопки и получают доступ к инструментам разработчика Neo4j, таким как браузер Neo4j, Neo4j Bloom и стандартная библиотека APOC. Существует два варианта Neo4j Aura: AuraDB и AuraDS. AuraDB – это стандартное предложение «база данных как услуга», которое подходит для поддержки веб-приложений и API. AuraDS – это платформа Neo4j для обработки графовых данных, включающая возможности, характерные для рабочих нагрузок по обработке и анализу данных. Мы будем использовать Neo4j AuraDB.

8.2.1. *Создание кластера Neo4j Aura*

Neo4j Aura – это управляемая служба, поэтому сначала нужно зарегистрироваться, войдя с учетной записью Google или создав учетную запись на основе адреса электронной почты и пароля на сайте **neo4j.com/aura** и выбрав **Sign Up Now** (Зарегистрироваться сейчас). Сам я использую Gmail, поэтому выберу вход с учетной записью Google. После входа появится панель управления Neo4j Aura.

Панель Neo4j Aura – это центр управления кластерами Neo4j в облаке. Она позволяет следить за существующими базами данных, создавать новые, импортировать данные, масштабировать базы данных вверх или вниз и пользоваться инструментами разработчика.

Однако, поскольку мы еще не создали ни одного кластера Neo4j Aura, панель управления выглядит пустой. Создадим новый кластер, щелкнув на кнопке **Create a database** (Создать базу данных), как показано на рис. 8.2. Существует тариф «AuraDB Free», предлагающий экземпляр Neo4j совершенно бесплатно, поэтому я выберу этот вариант. Для больших приложений можно выбрать тариф «AuraDB Professional», предлагающий дополнительные возможности и масштабирование.

Рис. 8.2. Настройка развертывания Neo4j AuraDB

Обязательно выбирайте тип базы данных *AuraDB Free*. Далее нужно выбрать имя для базы данных. Я выбрал название *GRANDstack Business Reviews*. Для развертывания базы данных доступно несколько регионов. Я просто оставил значение по умолчанию, но вы можете явно выбрать ближайшее к вам местоположение. В **Starting dataset** (Начальный набор данных) можно указать предопределенный набор данных или загрузить свои данные. Поскольку мы будем работать со своими данными, выберите **Load or create your own data in a blank database** (Загрузить или создать свои данные в пустой базе данных). После настройки параметров конфигурации будет предложен случайный пароль для доступа к экземпляру Neo4 Aura (рис. 8.3).

Рис. 8.3. Учетные данные для доступа к экземпляру Neo4j AuraDB

Обязательно сохраните пароль в надежном месте. Потом мы изменим его, но он понадобится нам для входа в браузер Neo4j.

Щелчок на кнопке **Continue** (Продолжить) вернет нас в панель управления Neo4j Aura, но теперь в ней появится информация о только что развернутом кластере базы данных с возможными вариантами действий **Explore** (Исследовать), **Query** (Запросить) или **Import** (Импортировать), как показано на рис. 8.4. Кнопка **Explore** (Исследовать) запускает браузер Neo4j, который мы использовали в предыдущих главах для выполнения запросов Cypher и визуализации результатов. Кнопка **Query** (Запросить) запускает Neo4j Bloom – инструмент визуального исследования графа. Мы рассмотрим его чуть ниже. Наконец, кнопка **Import** (Импортировать) запускает инструмент импортирования данных; он поможет вам выгрузить данные в Neo4j из простых файлов, например в формате CSV.

Рис. 8.4. В панели управления Neo4j AuraDB появилась наша новая база данных

Если щелкнуть на имени базы данных, на экране появятся подробная информация о ней и параметры настройки, в том числе:

- **Connection URI** (URI подключения) – строка для подключения к кластеру Neo4j с помощью клиентского драйвера Neo4j;
- **Tier** (тариф) – используемый тариф для этой базы данных (**Free** (Бесплатный), **Professional** (Профессиональный) или **Enterprise** (Корпоративный));
- **Cloud provider** (Облачный провайдер) – облачная платформа, на которой развернут кластер; в данном случае Google Cloud Platform;
- **Region** (Регион) – географический регион центра обработки данных, в котором развернут кластер;
- **Memory** (Память) – текущий размер базы данных, который можно увеличить или уменьшить в любое время.

Там же имеется раскрывающийся список **Open with** (Открыть с) для доступа к инструментам разработчика Neo4j Browser и Neo4j Bloom.

8.2.2. Подключение к кластеру Neo4j Aura

Теперь, подготовив кластер Neo4j Aura, можно попробовать подключиться к нему с помощью драйвера Neo4j JavaScript. Но прежде изменим начальный пароль для пользователя `neo4j` базы данных. Для этого запустим Neo4j Browser, с которым мы познакомились в предыдущих главах, щелкнув на кнопке **Query** (Запрос). Чтобы вспомнить, как пользоваться браузером Neo4j Browser, вернитесь к главе 3. После этого будет предложено выполнить вход под учетной записью пользователя `neo4j` с первоначальным назначенным паролем.

Чтобы изменить пароль, нужно выполнить запрос Cypher к базе данных *system*. Любые административные команды, такие как изменение паролей пользователей, должны посылаться этой системной базе данных. Сначала переключим Neo4j Browser на системную базу данных:

```
:use system
```

Затем используем команду `ALTER CURRENT USER`, чтобы изменить пароль по умолчанию пользователя `neo4j`:

```
ALTER CURRENT USER SET PASSWORD FROM
"<НАЧАЛЬНЫЙ_НАЗНАЧЕННЫЙ_ПАРОЛЬ>" TO "<НОВЫЙ_НАДЕЖНЫЙ_ПАРОЛЬ>"
```

Обязательно замените <НАЧАЛЬНЫЙ_НАЗНАЧЕННЫЙ_ПАРОЛЬ> первоначальным паролем, а <НОВЫЙ_НАДЕЖНЫЙ_ПАРОЛЬ> – новым надежным паролем. В оставшихся примерах мы будем использовать пароль *graphqlapi*, но вообще я советую применять более надежный пароль. Чтобы вернуться к базе данных *neo4j* по умолчанию, выполните команду `:use neo4j`.

> **ПРИМЕЧАНИЕ.** Такие команды, как :use, являются служебными, характерными для Neo4j Browser. Они не являются командами Cypher. Чтобы получить дополнительную информацию об этих командах, выполните в браузере Neo4j Browser команду `:help` или `:help commands`.

Изменив пароль пользователя базы данных, давайте проверим, сможем ли мы подключиться к нашему кластеру Neo4j Aura, используя драйвер Neo4j JavaScript. В панели управления Aura щелкните мышью на имени базы данных. В ответ должны появиться примеры кода, показывающие, как подключиться к экземпляру Neo4j Aura с помощью различных драйверов (рис. 8.5).

Рис. 8.5. На вкладке **Connect** (Подключение) в Neo4j Aura можно увидеть примеры программного кода на различных языках

В листинге 8.1 приводится адаптированная версия примера на JavaScript. Здесь он просто подсчитывает количество узлов в базе данных и возвращает результат. Создайте новый файл в каталоге API с именем aura-connect.js и скопируйте туда этот пример на JavaScript.

> **ПРИМЕЧАНИЕ.** Обратите внимание на схему `neo4j+s://` в URI. Выше мы использовали схему `bolt://`, определяющую подключение к конкретному экземпляру Neo4j. С помощью Neo4j Aura мы развернули кластер – группу экземпляров Neo4j, взаимодействующих друг с другом для репликации и распределения данных, – поэтому должны использовать схему `neo4j`, чтобы подсказать драйверу, что тот должен посылать запросы разным машинам в кластере. Знак + требует от драйвера использовать безопасное шифрованное соединение.

Листинг 8.1. aura-connect.js: отправка запроса экземпляру Neo4j Aura

```
(async () => {
  const neo4j = require("neo4j-driver");

  // исправьте следующую информацию о подключении в соответствии
  // с вашими настройками
  const uri = "neo4j+s://97a0fe69.databases.neo4j.io";
  const user = "neo4j";
  const password = "graphqlapi";

  const driver = neo4j.driver(uri, neo4j.auth.basic(user, password));
  const session = driver.session();

  try {
    const readQuery = `MATCH (n)
                       RETURN COUNT(n) AS num`;
    const readResult = await session.readTransaction((tx) =>
      tx.run(readQuery)
    );
    readResult.records.forEach((record) => {
      console.log(`Found ${record.get("num")} nodes in the database`);
    });
  } catch (error) {
    console.error("Something went wrong: ", error);
  } finally {
    await session.close();
  }

  await driver.close();
})();
```

Этот код импортирует драйвер Neo4j JavaScript, создает экземпляр драйвера с учетными данными для доступа к Neo4j Aura, выполняет запрос Cypher в транзакции для чтения, а затем выводит результаты в консоль. Запустив этот файл, мы

сможем убедиться в возможности подключения к нашей базе данных Neo4j Aura и что база данных в настоящее время пуста:

```
$ node aura-connect.js
Found 0 nodes in the database
```

Наш следующий шаг – выгрузить данные из локального экземпляра Neo4j, которые использовались для разработки приложения, в базу данных Neo4j Aura.

8.2.3. Выгрузка данных в Neo4j Aura

Ранее мы использовали команду `:play grandstack` в Neo4j Browser, чтобы загрузить некоторые данные для приложения, а теперь можем добавить информацию о пользователях, новые отзывы и компании. Давайте обсудим процесс получения дампа данных и его выгрузки из локальной базы данных Neo4j в новый кластер Neo4j Aura.

Есть несколько способов импортировать данные в Neo4j Aura, но мы используем инструмент push-to-cloud. Если выбрать вкладку **Import** (Импорт) в панели управления Neo4j Aura, появится интерфейс мастера, который проведет через этапы выгрузки локальной базы данных Neo4j в облачную базу данных Neo4j Aura. Мы пройдем эти этапы прямо сейчас.

Сначала убедимся, что локальная база данных Neo4j остановлена. Для этого щелкните на кнопке **Stop** (Стоп) в Neo4j Desktop (рис. 8.6).

Рис. 8.6. Остановка локальной базы данных и переход в панель управления

Затем откроем терминал в Neo4j Desktop, который позволит выполнить команду `neo4j-admin` для этого конкретного экземпляра Neo4j. Инструмент командной строки `neo4j-admin` поддерживает несколько полезных функций, таких как импорт больших объемов данных из файлов CSV, создание рекомендуемой конфигурации памяти и команду `push-to-cloud`, которую мы используем для выгрузки локальной базы данных в Neo4j Aura.

Выберите стрелку раскрывающегося списка рядом с кнопкой **Открыть** и выберите **Терминал**, чтобы открыть новое окно с командной строкой. Рабочий каталог для этой новой командной строки установлен в каталог, в котором был установлен этот конкретный экземпляр Neo4j:

```
$ pwd
/Users/lyonwj/Library/Application Support/com.Neo4j.Relate/Data/dbmss/
dbms-54c2c495-211d-408d-8c9e-6a65cce61d91
```

Теперь можно выполнить команду `push-to-cloud` для выгрузки этой базы данных в Neo4j Aura. Укажем Bolt URI нашего экземпляра Neo4j Aura, а также флаг `--overwrite`, чтобы стало ясно, что любые данные, которые могли быть созданы в экземпляре Neo4j Aura, нужно заменить. Нам будет предложено ввести имя пользователя и пароль, а затем запустится экспорт данных из локальной базы и их выгрузка в базу данных Neo4j Aura:

```
$ bin/neo4j-admin push-to-cloud --bolt-uri \
neo4j+s://97a0fe69.databases.neo4j.io --database neo4j --overwrite

Neo4j aura username (default: neo4j):neo4j
Neo4j aura password for neo4j:
Done: 68 files, 879.4KiB processed.
Dumped contents of database 'neo4j' into '/Users/lyonwj/Library/Application
Support/com.Neo4j.Relate/Data/dbmss/
dbms-54c2c495-211d-408d-8c9e-6a65cce61d91/dump-of-neo4j-1612960685687'
Upload
..................... 10%
..................... 20%
..................... 30%
..................... 40%
..................... 50%
..................... 60%
..................... 70%
..................... 80%
..................... 90%
..................... 100%
We have received your export and it is currently being loaded into
your Aura instance.
You can wait here, or abort this command and head over to the console to
```

```
be notified of when your database is running.
Import progress (estimated)
..................... 10%
..................... 20%
..................... 30%
..................... 40%
..................... 50%
..................... 60%
..................... 70%
..................... 80%
..................... 90%
..................... 100%
Your data was successfully pushed to Aura and is now running.
```

Теперь проверим, были ли выгружены данные в экземпляр Neo4j Aura. Если снова запустить сценарий aura-connect.js, он должен сообщить, что в базе данных имеется 36 узлов:

```
$ node aura-connect.js
Found 36 nodes in the database
```

8.2.4. Исследование графа с помощью Neo4j Bloom

Мы также можем визуально проверить и исследовать только что выгруженные данные. Вернемся к панели управления Neo4j Aura и откроем базу данных с помощью *Neo4j Bloom*. Neo4j Bloom – это приложение для визуального исследования графов Neo4j, включенное в состав Neo4j Aura. В панели управления Neo4j Aura щелкните на кнопке **Explore** (Исследовать). Откроется новая вкладка с предложением выполнить вход, используя имя пользователя и пароль базы данных.

После входа Neo4j Bloom подключится к экземпляру Neo4j Aura и позволит визуально исследовать данные в графе. Для начала давайте настроим *перспективу* (рис. 8.7). В Neo4j Bloom перспектива определяет способы представления графовых данных и как эти данные должны быть оформлены. Для наших целей вполне достаточно перспективы, созданной по умолчанию, поэтому выберите **Create Perspective** (Создать перспективу), чтобы создать перспективу для базы данных, а затем выберите созданную перспективу для визуализации.

После создания перспективы можно приступать к визуальному исследованию графа. Основное представление в Neo4j Bloom называется сценой и служит своего рода холстом, на котором можно рисовать отобранные графовые данные. Для вывода данных на сцену используется естественный язык, например в виде искомых строк в поле поиска, которые автоматически преобразуются в графовые шаблоны (рис. 8.8). Допустим, если начать вводить `User name: Will WROTE Review`, то нам начнут предлагаться подходящие графовые шаблоны для подстановки. Выбор одного из них приведет к выполнению поиска и заполнению сцены данными, соответствующими этому шаблону.

Рис. 8.7. Создание перспективы в Neo4j Bloom

Рис. 8.8. Поиск на естественном языке в Neo4j Bloom

Выше отмечалось, что перспективы позволяют настраивать стиль визуализации (рис. 8.9). Один из таких стилей, который можно настроить, – значки, используемые для представления узлов. Выбрав категорию в панели легенды, можно применить такие стили, как цвет, размер, значок или заголовок узла.

Визуализация имеет интерактивный характер и может использоваться для изучения графа или проверки соответствия выгруженных данных нашим ожиданиям. Выбирая узлы, можно просматривать их свойства (рис. 8.10). Также можно щелкнуть правой кнопкой мыши на узле или отношении, чтобы дополнительно развернуть либо отфильтровать данные, отображаемые в сцене.

Рис. 8.9. Настройка значков в Neo4j Bloom

На данный момент мы подготовили кластер Neo4j Aura, изменили пароль, выгрузили данные, исследовали эти данные, а также проверили и исследовали граф в Neo4j Bloom. Теперь перейдем к вопросу развертывания приложения React и GraphQL API с использованием Netlify и AWS Lambda.

Рис. 8.10. Просмотр дополнительной информации об узле в Neo4j Bloom

8.3. Развертывание приложения React с помощью Netlify Build

Чтобы развернуть наше приложение React, используем Netlify – облачную платформу, ориентированную на удобство создания, разработки и развертывания веб-приложений. Netlify включает автоматизированную систему сборки, глобальную сеть доставки контента, бессерверные функции, пограничные обработчики и другие возможности, объединенные в платформу, ориентированную на удобство работы разработчиков.

В числе других служб с аналогичными возможностями можно назвать: Vercel, DigitalOcean App Platform, Cloudflare Pages и Azure Static Web Apps. Служба Netlify предлагает также бесплатный тариф, поэтому можно развернуть и опробовать приложение без необходимости нести какие-либо расходы.

Netlify также позволяет запускать сборку и развертывать приложения после отправки изменений в систему управления версиями Git, например GitHub или GitLab. В этом разделе мы используем GitHub и покажем замечательную особенность Netlify – предварительные сборки, – позволяющую развертывать и тестировать приложение по запросу на включение (pull request).

8.3.1. *Добавление сайта в Netlify*

Для начала перейдем на сайт **https://www.netlify.com/** и щелкнем на кнопке **Sign up** (Зарегистрироваться), чтобы создать бесплатную учетную запись Netlify. Поскольку предполагается, что мы будем пользоваться преимуществами интеграции с GitHub для развертывания и обновления приложения из GitHub, можно войти в Netlify, используя учетную запись GitHub, и тогда ваша учетная запись Netlify будет связана с GitHub (рис. 8.11). Также можно войти с помощью электронной почты и пароля и связать учетную запись Netlify с GitHub позже.

Рис. 8.11. Регистрация в Netlify

После регистрации откроется обзорный список наших веб-сайтов, добавленных в Netlify (рис. 8.12). Так как мы еще ничего не сделали, только создали учетную запись, страница будет выглядеть довольно пустой.

Рис. 8.12. Панель управления Netlify

Чтобы добавить первый сайт в Netlify, создадим репозиторий GitHub для нашего приложения, который затем будем использовать для развертывания сайта в Netlify (рис. 8.13). Для этого откройте в веб-браузере страницу **https://github.com/new**, выберите название для репозитория – я выбрал *grandstack-business-reviews*. При желании репозиторий можно сделать приватным, чтобы никто, кроме нас, не имел к нему доступа.

После создания пустого репозитория GitHub в него нужно добавить код нашего приложения. На рис. 8.14 показаны типичные команды инициализации репозитория `git`, фиксации кода и отправки на GitHub. Существует также клиент с графическим интерфейсом; однако мы будем использовать командную строку.

Откройте терминал и перейдите в каталог `web-react`, где находится исходный код созданного нами приложения React. Первым шагом инициализируйте пустой репозиторий GitHub:

```
$ git init
```

Получить статус локального рабочего каталога можно командой `git status`:

```
$ git status

On branch main
```

```
No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    README.md
    package-lock.json
    package.json
    public/
    src/

nothing added to commit but untracked files present (use "git add" to track)
```

Рис. 8.13. Создание нового репозитория GitHub

Рис. 8.14. Инструкции по копированию кода из локального репозитория Git в новый репозиторий GitHub

На данный момент мы еще ничего не отправили в репозиторий, поэтому следующим шагом подготовим наш код к отправке. Для этого выполните команду `git add`:

```
$ git add -A
```

Флаг `-A` указывает, что мы хотим добавить все (All) файлы. Обычно в репозиторий добавляются *не все* файлы, например каталог `node_modules` и файлы с конфиденциальными данными не должны храниться в системе управления версиями. Инструмент `create-react-app`, который мы использовали для создания основы нашего приложения React, также создал файл `.gitignore`, описывающий, какие файлы не следует добавлять в `git`. Благодаря этому файлу можно без опаски использовать флаг `-A` при подготовке файлов к отправке. Теперь, если снова выполнить команду `git status`, она выведет список файлов, которые будут отправлены в репозиторий:

```
$ git status
On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file: .gitignore
    new file: README.md
    new file: package-lock.json
    new file: package.json
    new file: public/favicon.ico
    new file: public/index.html
```

```
    new file: public/logo192.png
    new file: public/logo512.png
    new file: public/manifest.json
    new file: public/robots.txt
    new file: src/App.css
    new file: src/App.js
    new file: src/App.test.js
    new file: src/BusinessResults.js
    new file: src/Profile.js
    new file: src/index.css
    new file: src/index.js
    new file: src/logo.svg
    new file: src/serviceWorker.js
    new file: src/setupTests.js
```

Давайте отправим наши файлы с помощью команды `git commit`. Каждая отправка (или фиксация) также включает сообщение, описывающее причину фиксации или новые возможности, добавленные в код. Сообщение можно добавить с помощью флага `-m`. Также можно опустить этот флаг и получить предложение ввести сообщение:

```
$ git commit -m "initial commit"
[main (root-commit) 0bb81ca] initial commit
  20 files changed, 14609 insertions(+)
  create mode 100644 .gitignore
  create mode 100644 README.md
  create mode 100644 package-lock.json
  create mode 100644 package.json
  create mode 100644 public/favicon.ico
  create mode 100644 public/index.html
  create mode 100644 public/logo192.png
  create mode 100644 public/logo512.png
  create mode 100644 public/manifest.json
  create mode 100644 public/robots.txt
  create mode 100644 src/App.css
  create mode 100644 src/App.js
  create mode 100644 src/App.test.js
  create mode 100644 src/BusinessResults.js
  create mode 100644 src/Profile.js
  create mode 100644 src/index.css
  create mode 100644 src/index.js
  create mode 100644 src/logo.svg
  create mode 100644 src/serviceWorker.js
  create mode 100644 src/setupTests.js
```

Далее подключим локальный репозиторий Git к удаленному репозиторию GitHub:

```
$ git remote add origin \
git@github.com:johnymontana/grandstack-business-reviews.git
```

И наконец, отправим локальную фиксацию в удаленный репозиторий командой `git push`:

```
$ git push -u origin main

Enumerating objects: 24, done.
Counting objects: 100% (24/24), done.
Delta compression using up to 16 threads
Compressing objects: 100% (24/24), done.
Writing objects: 100% (24/24), 175.60 KiB | 1.60 MiB/s, done.
Total 24 (delta 0), reused 0 (delta 0)
To github.com:johnymontana/grandstack-business-reviews.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Если после этого обновить веб-страницу GitHub с нашим репозиторием, то можно будет увидеть зафиксированный код и историю фиксаций (рис. 8.15).

Рис. 8.15. Страница репозитория GitHub после фиксации

Теперь можно приступать к развертыванию приложения React в Netlify. Вернитесь в панель управления Netlify и щелкните на кнопке **Add site from Git** (Добавить сайт из Git). Вам будет предложено выбрать поставщика услуг Git, к которому вы хотите подключиться, а затем, собственно, репозиторий. Выберите **GitHub** и только что созданный репозиторий, куда мы отправили наш код (рис. 8.16).

Рис. 8.16. Добавление нового сайта в Netlify

Netlify проверит код, определит, что он представляет приложение React, созданное с помощью команды `npm run build`, а контент должен обслуживаться из каталога /build. Здесь не нужно вносить никаких изменений, потому что для сборки и развертывания приложения React обычно достаточно настроек по умолчанию. При необходимости настройки сборки можно изменить позже (рис. 8.17).

Рис. 8.17. Настройка нового сайта в Netlify

Затем Netlify загрузит код из GitHub, соберет и развернет сайт. После этого можно заглянуть в журнал сборки в панели управления. Каждому сайту Netlify назначает URL и SSL-сертификат, поэтому мы можем сразу же просмотреть наше приложение без необходимости добавлять собственный домен (рис. 8.18).

Рис. 8.18. Настройка параметров развертывания в Netlify

Для этого откройте в веб-браузере страницу приложения (рис. 8.19). В данном случае я получил URL **https://hungry-thompson-86fbf3.netlify.app/**.

Рис. 8.19. Развернутый и действующий сайт в Netlify

Но есть одна проблема: ссылка на GraphQL API указывает на `http://localhost:4000` – на нашу локальную машину, а это означает, что любой другой, открывший это приложение, не сможет подключиться к GraphQL API и просмотреть результаты. Вы легко можете убедиться в этом, открыв инструменты разра-

ботчика в веб-браузере и проверив сетевые запросы (рис. 8.20). Мы развернем приложение GraphQL API в следующем разделе.

Рис. 8.20. Наше только что развернутое приложение

8.3.2. *Настройка переменных окружения для сборок Netlify*

Если заглянуть в src/index.js, где создается экземпляр Apollo Client для подключения к GraphQL API, мы увидим, что оставили URI-адрес GraphQL API `http://localhost:4000`, как показано в листинге 8.2.

Листинг 8.2. src/index.js: использование Apollo Link для подключения к GraphQL API

```
...
const httpLink = createHttpLink({
    uri: "http://localhost:4000",
});
...
```

Это обычное дело для локальной разработки и тестирования, но теперь было бы желательно использовать один и тот же код и для локальной разработки, и в развернутом приложении. Чтобы иметь возможность использовать URI локального GraphQL API во время разработки и развернутого GraphQL API в действующем приложении, установим переменную окружения, откуда будет извлекаться URI GraphQL API во время сборки. Мы определим это значение в зависимости от текущего окружения – для локальной разработки оставим URI равным `http://localhost:4000`, а для сборки в Netlify установим другое значение.

Итак, создадим файл .env с переменными окружения для локальной разработки. Одна из удобных особенностей Create React App – любые значения, указанные в файле .env, будут преобразованы в переменные окружения, а любые переменные с именами, начинающимися с `REACT_APP`, будут заменены их значениями в клиентском приложении React во время сборки. Давайте установим адрес GraphQL API для локальной разработки в этом файле .env, как показано в листинге 8.3.

Листинг 8.3. .env: настройка переменных окружения для приложения React

```
REACT_APP_GRAPHQL_URI=/graphql
NEO4J_URI=neo4j://localhost:7687
NEO4J_USER=neo4j
NEO4J_PASSWORD=letmein
REACT_APP_AUTH0_DOMAIN=grandstack.auth0.com
REACT_APP_AUTH0_CLIENT_ID=4xw3K3cjvw0hyT4Mjp4RuOVSxvVYcOFF
REACT_APP_AUTH0_AUDIENCE=https://reviews.grandstack.io
```

В листинге 8.4 показан обновленный код, читающий эти переменные окружения при настройке URI GraphQL API, домена Auth0, идентификатора клиента и аудитории.

Листинг 8.4. src/index.js: использование переменных окружения

```
...
const httpLink = createHttpLink({
    uri: process.env.REACT_APP_GRAPHQL_URI
});

...

ReactDOM.render(
  <React.StrictMode>
    <Auth0Provider
      domain={process.env.REACT_APP_AUTH0_DOMAIN}
      clientId={process.env.REACT_APP_AUTH0_CLIENT_ID}
      redirectUri={window.location.origin}
      audience={process.env.REACT_APP_AUTH0_AUDIENCE}
    >
      <AppWithApollo />
```

```
    </Auth0Provider>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Для локальной разработки должен использоваться локальный GraphQL API, а в действующем приложении – развернутый API GraphQL. Для этого установим переменную окружения `REACT_APP_GRAPHQL_URI` в настройках сборки нашего сайта на Netlify (рис. 8.21). Выберите **Site Settings** (Настройки сайта) в панели управления Netlify, а затем **Build & deploy** (Сборка и развертывание) слева, в панели навигации. Создайте новую переменную окружения с именем `REACT_APP_GRAPHQL_URI` и значением `/graphql`.

Рис. 8.21. Настройка переменных окружения в Netlify

В соответствии с этими настройками наше развернутое приложение попытается подключиться к GraphQL API по адресу `/graphql` в том же домене. Мы пока не развернули GraphQL API здесь, поэтому прямо сейчас наше приложение будет возвращать ошибку.

8.3.3. *Предварительное развертывание в Netlify*

Службы, подобные Netlify, предлагают удобную функцию – предварительный просмотр развертывания. *Предварительное развертывание* – это сборка, запускаемая после каждого изменения кода (часто по запросу на включение), которое развертывается по временному URL, отличному от URL основного приложения. Эта сборка обладает всеми возможностями основного приложения и может использоваться для ознакомления с тем, как отразятся изменения на основном приложении.

Давайте посмотрим, как это работает, создав запрос на включение и развернув предварительную версию нашего приложения React, читающую переменную окружения `REACT_APP_GRAPHQL_URI`. Запустив команду `git status`, можно увидеть, что мы внесли изменения в src/index.js:

```
$ git status

On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified: index.js

no changes added to commit (use "git add" and/or "git commit -a")
```

Переключимся на новую ветвь Git с именем `env-var-graphql-uri` и подтвердим изменения в этой новой ветви:

```
$ git checkout -b env-var-graphql-uri
Switched to a new branch 'env-var-graphql-uri'
```

Теперь отправим измененный файл index.js в эту ветвь. После переключения рабочего каталога в эту новую ветвь Git изменения будут отправлены в ветвь `env-var-graphql-uri`, а не в `main`:

```
$ git add index.js
$ git commit -m "use environment variable to specify GraphQL URI"
[env-var-graphql-uri 92f1142] use environment variable to
specify GraphQLURI
  1 file changed, 1 insertion(+), 1 deletion(-)
```

Далее отправим эту новую ветвь в репозиторий на GitHub. После этого GitHub сообщит, что можно создать запрос на включение (pull request) из этой новой ветви:

```
$ git push origin env-var-graphql-uri

Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.

Delta compression using up to 16 threads
```

```
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 415 bytes | 415.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'env-var-graphql-uri'on
GitHub by visiting: remote: https://github.com/johnymontana/
grandstack-business-reviews/pull/new/env-var-graphql-uri
remote:
To github.com:johnymontana/grandstack-business-reviews.git
  * [new branch]      env-var-graphql-uri -> env-var-graphql-uri
```

Запрос на включение – это способ запросить включение изменений из другой ветви или репозитория в основную ветвь `main`. Создадим запрос на включение, запрашивающий слияние новой ветви `env-var-graphql-uri` с основной ветвью (рис. 8.22).

Рис. 8.22. Создание запроса на включение в GitHub

Мы подключили Netlify к этому репозиторию GitHub, поэтому немедленно будет запущено развертывание предварительной сборки на основе изменений в этом запросе на включение. Увидеть статус сборки можно в разделе **Checks** (Проверки) на странице запроса на включение на GitHub. После завершения сборки можно открыть эту предварительную версию, чтобы увидеть, как отразились изменения на поведении приложения (рис. 8.23). Этот временный URL можно также передать другим заинтересованным лицам, чтобы они могли увидеть и прокомментировать изменения на сайте.

Рис. 8.23. Запуск развертывания предварительной сборки на Netlify по запросу на включение

Если все в порядке, то можно выполнить слияние с основной ветвью. Для этого достаточно щелкнуть на кнопке **Merge pull request** (Принять запрос на включение) на GitHub. После этого изменения из ветви `env-var-graphql-uri` будут включены в основную ветвь. Это слияние, в свою очередь, запустит сборку и развертывание новой версии приложения в Netify, которая затем заменит текущую (рис. 8.24).

Теперь, развернув приложение React, перейдем к развертыванию GraphQL API. Для этого преобразуем GraphQL API в бессерверную функцию, чтобы ее можно было развернуть в AWS Lambda. Используем функцию Netlify Functions.

Рис. 8.24. Состояние сборки в Netlify

8.4. Развертывание GraphQL в виде бессерверной функции с помощью AWS Lambda и Netlify Functions

AWS Lambda – это вычислительная платформа FaaS (Function as a Service – функция как услуга), позволяющая запускать код по запросу без выделения серверов или управления ими. Функции вызываются в ответ на события, такие как HTTP-запрос. В сочетании со службой AWS Gateway API бессерверные функции можно использовать для реализации конечных точек API и приложений, таких как GraphQL API. AWS Lambda поддерживает Node.js, Python, Java, Go, Ruby, Swift и C# и может подключать упакованные зависимости. В отличие от других облачных служб, расходы на которые измеряются по часам, стоимость AWS Lambda зависит от количества запросов, а продолжительность этих запросов измеряется с шагом в 1 миллисекунду.

Служба Netlify Functions позволяет развертывать функции Lambda непосредственно из функций Netlify без создания учетной записи AWS. Netlify выполняет сборку и развертывание функций Lambda, используя те же возможности управления версиями Git, такие как предварительное развертывание, соответственно, мы можем управлять кодом наших функций Lambda вместе с остальной частью сайта. В настоящее время Netlify поддерживает развертывание функций Lambda на Node.js и Go.

До сих пор мы создавали GraphQL API как сервер Node.js Express, используя Apollo Server. В этом разделе мы преобразуем наш GraphQL API в функцию Lambda, используя версию Apollo Server, предназначенную специально для Lambda, и развернем вместе с нашим сайтом Netlify с помощью Netlify Functions.

8.4.1. *GraphQL API в виде бессерверной функции*

Наш Lambda GraphQL API будет развернут посредством Netlify как часть сайта, поэтому добавим код и зависимости в существующий проект. Для начала установим необходимые зависимости:

```
npm install apollo-server-lambda @neo4j/graphql
➥ @neo4j/graphql-plugin-auth neo4j-driver
```

Обратите внимание: эта команда устанавливает `apollo-server-lambda` – специализированную версию Apollo Server, которая позволит нам преобразовать наш GraphQL API в функцию Lambda. Также она устанавливает драйвер Neo4j JavaScript, библиотеку интеграции Neo4j GraphQL и библиотеки, необходимые для работы с JWT, которые мы использовали в предыдущей главе.

Создадим новый файл src/graphql.js в том же каталоге, где находится приложение React. Позже мы поместим этот файл в систему управления версиями и отправим его на GitHub, запустив сборку и развертывание Netlify. Мы используем `apollo-server-lambda` для создания простого GraphQL API с одним запросом `greetings`, возвращающим приветственное сообщение (листинг 8.5).

Листинг 8.5. src/graphql.js: простой GraphQL API на основе AWS Lambda

```
const { ApolloServer, gql } = require("apollo-server-lambda");

const typeDefs = gql`
  type Query {
    greetings(name: String = "GRANDstack"): String
  }
`;

const resolvers = {
  Query: {
    greetings: (parent, args, context) => {
      return `Hello, ${args.name}!`;
    },
  },
};

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

const serverHandler = server.createHandler();

exports.handler = (event, context, callback) => {
  return serverHandler(
    {
      ...event,
      requestContext: event.requestContext || {},
    },
    context,
    callback
  );
};
```

Обратите внимание: здесь импортируется `apollo-server-lambda` **– специализированная версия Apollo Server**

Чтобы создать функцию AWS Lambda, нужно экспортировать функцию-обработчик, которая является оберткой для нашего экземпляра Apollo Server

Затем нам нужно настроить сайт Netlify, чтобы он знал, где находится наша новая функция Lambda и что она должна быть доступна как конечная точка `/graphql` нашего сайта. Для этого создадим файл netlify.toml в корне проекта (листинг 8.6).

Листинг 8.6. netlify.toml: настройка сборки в Netlify

```
[build]
command = "npm run build"
functions = "src/lambda"
```

```
publish = "build"

[[redirects]]
from = "/graphql"
to = "/.netlify/functions/graphql"
status = 200
```

По умолчанию функции Netlify доступны по адресу `/.netlify/functions/`, за которым следует имя файла функции. Мы определили перенаправление, поэтому наш GraphQL API будет доступен также по адресу /graphql.

8.4.2. *dev: интерфейс командной строки Netlify*

До сих пор мы рассматривали Netlify как службу развертывания приложений React. Когда мы собирали и поддерживали локальное приложение React, то использовали команду `npm run start` и инструмент `react-scripts` без привлечения Netlify. Теперь, после добавления функции Lambda, нужно сделать еще один шаг, чтобы получить возможность тестировать приложение локально. Для этого установим инструмент командной строки Netlify, с помощью которого будем собирать и запускать нашу функцию Lambda GraphQL и приложение React локально с помощью Netlify dev:

```
$ npm install netlify-cli -g
```

Теперь можно использовать команду `dev` для запуска нашего сайта на локальном компьютере. Эта команда соберет и запустит наше приложение React и функцию Lambda локально, не запуская развертывание:

```
$ netlify dev
```

После запуска `netlify dev` можно запустить веб-браузер и ввести адрес `http://localhost:8888/graphql`. В ответ должен появиться веб-интерфейс Apollo Studio, в котором можно запустить запрос GraphQL к Lambda GraphQL API (листинг 8.7).

Листинг 8.7. Простой запрос к GraphQL API

```
{
  greetings
}
```

Этот запрос вернет приветственное сообщение, которое определено в функции разрешения:

```
{
  "data": {
    "greetings": "Hello, GRANDstack!"
  }
}
```

Конечно, это всего лишь вариант «Hello World» для GraphQL API. Поэтому не будем останавливаться на достигнутом и перенесем остальную часть GraphQL API для нашего приложения.

8.4.3. Преобразование GraphQL API в функцию Netlify

Как показано в листинге 8.8, чтобы преобразовать существующий GraphQL API для использования AWS Lambda и `apollo-server-lambda`, нужно изменить всего несколько строк. Наиболее значительные изменения касаются импорта пакета `apollo-server-lambda` вместо `apollo-server-express` и экспорта функции-обработчика AWS Lambda. Остальной код останется похожим на код GraphQL API, созданный в главе 7.

Листинг 8.8. src/graphql.js: преобразование GraphQL API в функцию AWS Lambda

```
const { ApolloServer, gql } = require("apollo-server-lambda");

const neo4j = require("neo4j-driver");
const { Neo4jGraphQL } = require("@neo4j/graphql");
const {
  Neo4jGraphQLAuthJWKSPlugin,
} = require("@neo4j/graphql-plugin-auth");

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

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

  type Business {
    businessId: ID!
```

Вместо apollo-server используется `apollo-server-lambda` – вариант Apollo Server

```
  waitTime: Int! @computed
  averageStars: Float!
    @auth(rules: [{ isAuthenticated: true }])
    @cypher(
      statement: """
        MATCH (this)<-[:REVIEWS]-(r:Review) RETURN avg(r.stars)
      """
    )

  recommended(first: Int = 1): [Business!]!
    @cypher(
    statement: """
    MATCH (this)<-[:REVIEWS]-(:Review)<-[:WROTE]-(u:User)
    MATCH (u)-[: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!]! @relationship(type: "REVIEWS", direction: IN)
  categories: [Category!]!
    @relationship(type: "IN_CATEGORY", direction: OUT)
}

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

extend type User
  @auth(
    rules: [
      { operations: [READ], where: { userId: "$jwt.sub" } }
      { operations: [CREATE, UPDATE, DELETE], roles: ["admin"] }
    ]
  )

type Review {
  reviewId: ID! @id
  stars: Float!
  date: Date!
```

```
    text: String
    user: User! @relationship(type: "WROTE", direction: IN)
    business: Business! @relationship(type: "REVIEWS", direction: OUT)
  }

  extend type Review
    @auth(
      rules: [
        {
          operations: [CREATE, UPDATE]
          bind: { user: { userId: "$jwt.sub" } }
        }
      ]
    )

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

const driver = neo4j.driver(
  process.env.NEO4J_URI,
  neo4j.auth.basic(process.env.NEO4J_USER, process.env.NEO4J_PASSWORD)
);

const neoSchema = new Neo4jGraphQL({
  typeDefs,
  resolvers,
  driver,
  plugins: {
    auth: new Neo4jGraphQLAuthJWKSPlugin({
      jwksEndpoint: "https://grandstack.auth0.com/.well-known/jwks.json",
    }),
  },
});

const initServer = async () => {
  return await neoSchema.getSchema().then((schema) => {
    const server = new ApolloServer({
      schema,
      context: ({ event }) => ({ req: event }),
    });
    const serverHandler = server.createHandler();
    return serverHandler;
```

Здесь используется объект event, **потому что сигнатура запроса для AWS Lambda немного отличается от Express**

```
  });
};

exports.handler = async (event, context, callback) => {
  const serverHandler = await initServer();

  return serverHandler(
    {
      ...event,
      requestContext: event.requestContext || {},
    },
    context,
    callback
  );
};
```

Экспорт функции-обработчика для доступа к функции AWS Lambda

Теперь мы можем зафиксировать изменения в этом файле и отправить их на GitHub для развертывания. Осталось только добавить собственный домен и назначить его нашему сайту в Netlify, чем мы и займемся в следующем разделе.

8.4.4. Добавление собственного домена в Netlify

До сих пор наше приложение работало в поддомене **https://hungry-thompson-86fbf3.netlify.app/**, автоматически назначенном платформой Netlify. Давайте определим свой домен, лучше соответствующий бренду нашего сайта. В Netlify выберите раздел **Domains** (Домены) в верхней панели навигации. Здесь можно добавлять свои домены и назначать их сайтам в Netlify (рис. 8.25).

Рис. 8.25. Добавление своего домена в Netlify

Приобрести доменное имя можно непосредственно у Netlify или использовать домен, приобретенный у другого регистратора. В этом примере я добавлю домен, купленный у другого регистратора, поэтому настрою его на серверах имен Netlify, что позволит платформе Netlify управлять доменом и записями DNS (рис. 8.26).

Рис. 8.26. Настройка домена на серверах имен Netlify

Наконец, нужно обновить настройки приложения Auth0, чтобы функции аутентификации, предоставляемые Auth0, использовали новый домен. Для этого изменим **Allow Callback URLs** (Допустимые URL обратных вызовов) и **Allowed Logout URLs** (Допустимые URL выхода) в Auth0, указав URL локального хоста по умолчанию, а также URL нашего сайта Netlify и cdjq-домен (рис. 8.27).

Рис. 8.27. Настроенные допустимые URL для обратных вызовов в Auth0

Теперь наше приложение будет развернуто и готово к использованию в нашем домене (рис. 8.28).

Рис. 8.28. Вид приложения GraphQL после входа

8.5. Наш подход к развертыванию

В этой главе мы рассмотрели подход к развертыванию приложения GraphQL, в котором использовались преимущества управляемых служб, в частности Neo4j Aura, Netlify и AWS Lambda (рис. 8.29). В начале этой главы перечислялись некоторые преимущества и недостатки управляемых служб в целом. Теперь взглянем на них с точки зрения разработчика.

Рис. 8.29. Развертывание приложения GraphQL с точки зрения разработчика

Netlify позволяет автоматически собирать и развертывать приложения React в глобальной сети доставки контента Netlify, гарантируя его доступность всему миру без лишних сетевых задержек. Преобразование GraphQL API в функцию AWS Lambda и использование Netlify Functions позволяет интегрировать прикладной API в единую базу кода. Благодаря интеграции с GitHub рабочий процесс разработки и развертывания существенно упростился, к тому же мы получили возможность создавать предварительные сборки по запросам на включение.

База данных Neo4j Aura как услуга позволяет использовать инструменты разработчика, такие как Neo4j Desktop и Neo4j Browser, для разработки без необходимости беспокоиться о поддержке и эксплуатации кластера Neo4j в облаке. Теперь, развернув приложение, в следующей главе мы отложим его в сторону и поговорим о более мощных возможностях GraphQL, таких как абстрактные типы, разбивка на страницы на основе курсора и модель соединения Relay, а также операции со свойствами отношений в графе.

8.6. Упражнения

1. Используйте Neo4j Bloom и отыщите пользователя, оставившего отзывы к компаниям из наибольшего количества категорий. Для компаний из каких категорий оставил отзывы этот пользователь? Подсказка: выполнить это упражнение может помочь создание поисковой фразы Neo4j Bloom. За дополнительной информацией обращайтесь к документации, доступной по адресу **http://mng.bz/XZR6**.
2. Создайте новый запрос на включение в приложение поддержки упорядочивания всех результатов по названиям компаний. Используйте поддержку развертывания в Netlify, чтобы просмотреть это обновление перед слиянием запроса на включение и обновлением приложения.
3. Создайте новую функцию Netlify, использующую драйвер Neo4j JavaScript для отправки запроса в кластер Neo4j Aura и получения списка самых последних отзывов. Выполните его локально с помощью команды `netlify dev` перед развертыванием. Используйте конфигурацию netlify.toml для перенаправления /reviews на эту функцию.

Итоги

- Использование управляемых облачных служб может упростить процесс развертывания и обслуживания веб-приложений, позволяет задать свой адрес и предлагает цены, которые могут быть привлекательными для разработчиков, несущих ответственность за все компоненты приложения.
- Neo4j Aura – это управляемая облачная служба баз данных, предоставляющая кластеры Neo4j, которые можно инициализировать одним щелчком мыши. Эти экземпляры автоматически масштабируются вверх и вниз по мере необходимости, что устраняет необходимость в обслуживании или поддержке Neo4j.
- Платформу Netlify и ее сеть доставки контента можно использовать для автоматизации сборки и развертывания веб-приложений, получения пре-

имуществ интеграции с GitHub и развертывания предварительных версий, помогающих увидеть влияние изменений перед их отправкой.

- GraphQL API можно развернуть как функцию AWS Lambda, используя преимущества масштабирования и ценообразования на основе загруженности, которые добавляют привлекательности AWS Lambda. Netlify Functions можно использовать для организации частей сайта Netlify в виде функций AWS Lambda, что устраняет необходимость в поддержке отдельной базы кода или процесса развертывания.

Глава 9

Продвинутые возможности GraphQL

В этой главе:

- абстрактные типы объединений и типы интерфейсов;
- разбиение результатов запроса на страницы с использованием смещений и курсоров;
- работа со свойствами отношений с применением типов соединения Relay.

Мы еще не использовали одну из самых мощных и важных особенностей системы типов GraphQL – *абстрактные типы*, позволяющие представлять несколько конкретных типов в одном поле GraphQL. Точно так же мы не использовали *свойства отношений* – важнейшую особенность графовой модели данных, – которые позволяют придавать атрибуты не только узлам, но и отношениям, связывающим узлы. В этой главе я расскажу, как использовать абстрактное объединение и типы интерфейсов, поддерживаемые в GraphQL, а также свойства отношений. Попутно я представлю объекты GraphQL Connection и методы разбиения результатов на страницы. Мы оставим в стороне наше приложение обзора компаний и упростим модель данных, сосредоточившись на API простого интернет-магазина, продающего два типа продуктов: книги и видео.

9.1. Абстрактные типы GraphQL

GraphQL поддерживает два вида абстрактных типов: *интерфейсы* и *объединения*. Абстрактные типы позволяют представлять несколько конкретных типов (или массивы нескольких типов) в одном поле. Интерфейсы используются, когда несколько конкретных типов имеют одно или несколько одинаковых полей, и объявляют общие поля, которые должны быть реализованы в конкретном типе. Проще говоря, интерфейс можно рассматривать как контракт, определяющий минимальный набор полей в типе, чтобы тот считался реализующим указанный

интерфейс. Конкретные типы, входящие в объединения, не обязаны иметь общие и не поддерживают эту идею контракта. Таким образом, объединения – это просто группы конкретных типов.

9.1.1. *Интерфейсы*

Интерфейсы используются для представления концептуально схожих типов объектов и имеющих хотя бы одно общее поле. Например, в API интернет-магазина может поддерживаться понятие *человек*. Человек может быть клиентом и сотрудником. Объект, представляющий человека, будет иметь такие поля, как имя, фамилия и имя пользователя. Однако только у клиента будет адрес доставки, и только у сотрудника будет дата приема на работу. В определениях типов GraphQL эту концепцию можно представить, как показано в листинге 9.1.

Листинг 9.1. Определение и использование интерфейса в GraphQL

```
interface Person {
  firstName: String!
  lastName: String!
  username: String!
}

type Customer implements Person {
  firstName: String!
  lastName: String!
  username: String!
  shippingAddress: String
}

type Employee implements Person {
  firstName: String!
  lastName: String!
  username: String!
  hireDate: DateTime!
}

type Query {
  people: [Person]
}
```

Реализующий (конкретный) тип должен объявить все поля, присутствующие в интерфейсе, и может также объявлять другие поля, присущие только этому конкретному типу. В этом примере оба типа, `Customer` и `Employee`, реализуют интерфейс `Person` и, следовательно, должны включать поля `firstName`, `lastName` и `username`. Тип `Customer` добавляет поле `shippingAddress`, а тип `Employee` – поле `hireDate`.

В поле запроса `people` будет возвращаться массив объектов, каждый объект в котором может быть объектом `Employee` или `Customer`. Чтобы указать набор выбираемых полей для каждого конкретного типа, в запросах используются *встроенные*

фрагменты (листинг 9.2). Встроенные фрагменты позволяют запрашивать поля конкретного типа и проверять соответствие тому или иному конкретному типу.

Листинг 9.2. Запрос с применением интерфейса и встроенных фрагментов

```
{
  people {
    __typename
    firstName
    lastName
    username
    ... on Customer {
      shippingAddress
    }
    ... on Employee {
      hireDate
    }
  }
}
```

Здесь в выбираемом наборе полей присутствует метаполе `__typename`, через которое передается конкретный тип каждого объекта в массиве `people`.

9.1.2. Объединения

Объединения похожи на интерфейсы. Они тоже являются абстрактными типами, которые можно использовать для представления нескольких конкретных типов; однако конкретные типы, входящие в состав объединения, необязательно должны иметь общие поля. Обычно объединения используются для представления результатов поиска. Например, API нашего магазина может поддерживать функцию поиска товаров, которые могут быть книгами или видео. Для этой цели можно создать объединение `Product`, содержащее типы `Book` и `Video`, и поле запроса `search`, возвращающее массив объектов `Product` (листинг 9.3).

Листинг 9.3. Определение объединения в GraphQL

```
type Video {
  name: String!
  sku: String!
}

type Book {
  title: String!
  isbn: String!
}

union Product = Video | Book

type Query {
```

```
  search(term: String!): [Product!]!
}
```

По аналогии с тем, как с помощью встроенного фрагмента запрашиваются поля конкретных типов, реализующих интерфейс, при запросе объединений тоже можно использовать встроенные фрагменты. Однако, поскольку сам тип объединения не содержит никаких полей, при запросе объединения без использования встроенного фрагмента можно запросить только метаполе `__typename` (листинг 9.4).

Листинг 9.4. Запрос объединения

```
{
  search(term: "GraphQL") {
    __typename
    ... on Book {
      title
      isbn
    }
    ... on Video {
      name
      sku
    }
  }
}
```

9.1.3. *Использование абстрактных типов с библиотекой Neo4j GraphQL*

Теперь, получив первое представление об интерфейсах и объединениях, посмотрим, как можно использовать абстрактные типы в GraphQL API с помощью библиотеки Neo4j GraphQL. Отложим в сторону наше приложение обзора компаний и начнем новое приложение для воображаемого интернет-магазина, продающего книги и видео. Создайте новый каталог, перейдите в него и выполните следующую команду, чтобы создать новый проект Node.js:

```
npm init -y
```

Далее установим зависимости для нового приложения Node.js GraphQL API, которые уже должны быть вам знакомы:

```
npm install @neo4j/graphql graphql apollo-server neo4j-driver dotenv
```

Если позднее вы хотите продолжить эксперименты с приложением обзора компаний, созданным в предыдущих главах, то создайте новую базу данных Neo4j в Neo4j Aura или локально, используя Neo4j Desktop. В противном случае можно продолжить использовать существующую базу данных и выполнить следующую инструкцию Cypher, удаляющую данные для приложения обзора компаний:

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

Создайте новый файл .env, чтобы определить переменные окружения `NEO4J_USER`, `NEO4J_URI` и `NEO4J_PASSWORD` с учетными данными для подключения к базе данных Neo4j, как показано в листинге 9.5.

Листинг 9.5. .env: не забудьте подставить свои учетные данные для подключения к Aura

```
NEO4J_URI=neo4j+s://932a071e.databases.neo4j.io
NEO4J_USER=neo4j
NEO4J_PASSWORD=wH4-tvNOxzKlDZwIEqgNPm-8iS-tJ9gOgr1ScSq9yiM
```

Теперь, создав новый проект Node.js и имея новую или пустую базу данных Neo4j, перейдем к определению типов GraphQL для нашего API и посмотрим, как абстрактные типы могут помочь упростить схему.

Моделирование API интернет-магазина

Начнем создание нового API с перехода на (виртуальную) доску: **https://arrows.app**. Следуя процессу моделирования графовых данных, описанному в главе 3, определим объекты (узлы), связи (отношения) между ними и их атрибуты (свойства узлов). Не будем особенно усложнять и сосредоточимся на пользователях, которые будут размещать заказы на приобретение книг и/или видео. Этим требованиям соответствует довольно простая графовая модель (рис. 9.1).

Рис. 9.1. Графовая модель данных для интернет-магазина, торгующего книгами и видео

Как было показано в главе 4, такие диаграммы можно использовать для определения типов GraphQL, соответствующих этой графовой модели, применяя директиву схемы GraphQL `@relationship` для фиксации направлений и типов отношений, как показано в листинге 9.6.

Листинг 9.6. Определения типов GraphQL для модели данных интернет-магазина

```
type User {
  username: String
  orders: [Order!]! @relationship(type: "PLACED", direction: OUT)
```

```
}

type Order {
  orderId: ID! @id
  created: DateTime! @timestamp(operations: [CREATE])
  customer: User! @relationship(type: "PLACED", direction: IN)
  books: [Book!]! @relationship(type: "CONTAINS", direction: OUT)
  videos: [Video!]! @relationship(type: "CONTAINS", direction: OUT)
}

type Video {
  name: String
  sku: String
}

type Book {
  title: String
  isbn: String
}
```

Обратите внимание на директивы `@id` и `@timestamp`, используемые для автоматического создания соответствующих значений, – они избавляют клиента от необходимости передавать их в API. Наш клиент не должен заботиться о создании случайного уникального идентификатора для заказа или о передаче времени создания заказа, так как это может привести к проблемам с безопасностью.

Но взгляните на поля `Order.books` и `Order.videos`. Чтобы увидеть, какие товары содержатся в заказе, клиенту потребуется запросить оба этих поля, одно из которых может быть пустым массивом. Это немного неудобно для клиента; давайте посмотрим, как можно исправить ситуацию с помощью абстрактных типов, например объединений, поскольку наши типы `Video` и `Book` не имеют общих полей. Вместо полей `Order.books` и `Order.videos` определим новый тип объединения `Product` (листинг 9.7) и добавим поле `Order.products`, которое позволит работать с товарами в заказе (будь то книги или видео), используя одно поле.

Листинг 9.7. Определения типов GraphQL для модели данных интернет-магазина с использованием объединения

```
type User {
  username: String
  orders: [Order!]! @relationship(type: "PLACED", direction: OUT)
}

union Product = Video | Book

type Order {
  orderId: ID! @id
  created: DateTime! @timestamp(operations: [CREATE])
```

Определение объединения с именем `Product`, которое может представлять объекты типов `Video` и `Book`

```
  customer: User! @relationship(type: "PLACED", direction: IN)
  products: [Product!]! @relationship(type: "CONTAINS", direction: OUT)
}

type Video {
  name: String
  sku: String
}

type Book {
  title: String
  isbn: String
}
```

Использование нового типа `Product` **в поле отношения в типе** `Order`

Создание сервера GraphQL

Теперь, завершив определение типов GraphQL, воспользуемся ими для создания GraphQL API с помощью библиотеки Neo4j GraphQL. Создадим новый файл index.js с этими новыми определениями типов и добавим в него код, необходимый для создания GraphQL API с помощью Apollo Server и библиотеки Neo4j GraphQL (листинг 9.8).

Листинг 9.8. index.js: GraphQL API для интернет-магазина

```
const { gql, ApolloServer } = require("apollo-server");
const { Neo4jGraphQL } = require("@neo4j/graphql");
const neo4j = require("neo4j-driver");
require("dotenv").config();

const typeDefs = gql`
  type User {
    username: String
    orders: [Order!]! @relationship(type: "PLACED", direction: OUT)
  }

  union Product = Video | Book

  type Order {
    orderId: ID! @id
    created: DateTime! @timestamp(operations: [CREATE])
    customer: User! @relationship(type: "PLACED", direction: IN)
    products: [Product!]!
    @relationship(
      type: "CONTAINS"
```

```
      direction: OUT
    )
  }

  type Video {
    name: String
    sku: String
  }

  type Book {
    title: String
    isbn: String
  }
`;

const driver = neo4j.driver(
  process.env.NEO4J_URI,
  neo4j.auth.basic(process.env.NEO4J_USER, process.env.NEO4J_PASSWORD)
);

const neoSchema = new Neo4jGraphQL({ typeDefs, driver });

neoSchema.getSchema().then((schema) => {
  const server = new ApolloServer({
    schema,
  });
  server.listen().then(({ url }) => {
    console.log(`GraphQL server ready on ${url}`);
  });
});
```

Структура этого файла должна быть вам знакома по прошлым главам, где мы определяли типы GraphQL, создавали экземпляр драйвера Neo4j и генерировали схему GraphQL с помощью библиотеки Neo4j GraphQL. Теперь запустим наш сервер GraphQL:

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

Использование абстрактных типов в мутациях GraphQL

Запустите веб-браузер и введите адрес **http://localhost:4000**, чтобы открыть страницу Apollo Studio. Сейчас мы создадим некоторые данные в базе данных, используя мутации GraphQL, сгенерированные библиотекой Neo4j GraphQL в нашей схеме. Для начала создадим две учетные записи пользователей, используя сгенерированную мутацию `createUsers` (листинг 9.9).

Листинг 9.9. Мутация GraphQL: создание учетных записей пользователей

```
mutation {
  createUsers(
    input: [{ username: "bobbytables" }, { username: "graphlover123" }]
  ) {
    users {
      username
    }
  }
}
```

В ответ на эту операцию GraphQL должны появиться объекты `User` с именами пользователей, переданными в операцию мутации:

```
{
  "data": {
    "createUsers": {
      "users": [
        {
          "username": "bobbytables"
        },
        {
          "username": "graphlover123"
        }
      ]
    }
  }
}
```

Затем создадим несколько товаров в нашем магазине. Для этого используем мутации `createBooks` и `createVideos` (листинг 9.10).

Листинг 9.10. Мутация GraphQL: создание товаров

```
mutation {
  createBooks(
    input: [
      { title: "Full Stack GraphQL", isbn: "9781617297038" }
      { title: "Graph Algorithms", isbn: "9781492047681" }
      { title: "Graph-Powered Machine Learning", isbn: "9781617295645" }
    ]
  ) {
    books {
      title
```

```
      isbn
    }
  }

  createVideos(
    input: [
      { name: "Intro To Neo4j 4.x", sku: "v001" }
      { name: "Building GraphQL APIs", sku: "v002" }
    ]
  ) {
    videos {
      sku
      name
    }
  }
}
```

В результате в базе данных появятся массивы с объектами `Book` и `Video`:

```
{
  "data": {
    "createBooks": {
      "books": [
        {
          "title": "Full Stack GraphQL",
          "isbn": "9781617297038"
        },
        {
          "title": "Graph Algorithms",
          "isbn": "9781492047681"
        },
        {
          "title": "Graph-Powered Machine Learning",
          "isbn": "9781617295645"
        }
      ]
    },
    "createVideos": {
      "videos": [
        {
          "sku": "v001",
          "name": "Intro To Neo4j 4.x"
        },
        {
          "sku": "v002",
```

```
          "name": "Building GraphQL APIs"
        }
      ]
    }
  }
}
```

Теперь можно начинать создавать заказы. Сделать это можно несколькими способами, например с помощью мутации `updateUsers`, но давайте воспользуемся мутацией `createOrders`, как показано в листинге 9.11. Поскольку значения для полей `created` и `orderId` генерируются автоматически, нам не нужно указывать эти значения в мутации.

Листинг 9.11. Мутация GraphQL: создание одного заказа

```
mutation {
  createOrders(
    input: {
      customer: {
        connect: { where: { node: { username: "graphlover123" } } }
      }
      products: {
        Book: {
          connect: [
            { where: { node: { title: "Graph Algorithms" } } }
            { where: { node: { title: "Full Stack GraphQL" } } }
          ]
        }
        Video: {
          connect: { where: { node: { name: "Building GraphQL APIs" } } }
        }
      }
    }
  ) {
    orders {
      orderId
      created
      customer {
        username
      }
      products {
        __typename
        ... on Book {
          title
          isbn
```

```
        }
        ... on Video {
          name
          sku
        }
      }
    }
  }
}
```

Обратите внимание на встроенные фрагменты, используемые в поле `products`. Мы знаем, что это поле возвращает массив объектов `Product` типа объединения, каждый из которых может быть объектом `Book` или `Video`. Мы можем добавить в выборку поле `__typename`, которое сообщит нам конкретный тип каждого объекта, но, чтобы вернуть фактические поля конкретного типа (`Book` или `Video`), нужно использовать встроенный фрагмент, дабы указать поля, которые должны возвращаться, когда конкретный тип объекта соответствует типу, указанному во встроенном фрагменте:

```
products {
  __typename
  ... on Book {
    title
    isbn
  }
  ... on Video {
    name
    sku
  }
}
```

В полученном объекте ответа вы увидите, что нашим объектам `Order` присваиваются случайное значение идентификатора и отметка времени. Обратите внимание, что массив `products` содержит смесь объектов `Book` и `Video`:

```
{
  "data": {
    "createOrders": {
      "orders": [
        {
          "orderId": "dfdebf08-3ce5-494e-9843-d5286f4dc8f4",
          "created": "2021-08-15T13:43:15.117Z",
          "customer": {
            "username": "graphlover123"
          },
```

```
          "products": [
            {
              "__typename": "Video",
              "name": "Building GraphQL APIs",
              "sku": "v002"
            },
            {
              "__typename": "Book",
              "title": "Graph Algorithms",
              "isbn": "9781492047681"
            },
            {
              "__typename": "Book",
              "title": "Full Stack GraphQL",
              "isbn": "9781617297038"
            }
          ]
        }
      ]
    }
  }
}
```

Если открыть Neo4j Browser и проверить данные, созданные с помощью нашего GraphQL API, то можно увидеть графовое представление нашего заказа, пользователей и товаров, а также отношения между ними (рис. 9.2).

Рис. 9.2. Заказ содержит две книги и одно видео

Теперь создадим еще несколько заказов, используя другую мутацию GraphQL (листинг 9.12).

Листинг 9.12. Мутация GraphQL: создание нескольких заказов

```
mutation {
  createOrders(
    input: [
      {
        customer: {
          connect: { where: { node: { username: "bobbytables" } } }
        }
        products: {
          Book: {
            connect: { where: { node: { isbn: "9781617297038" } } }
          }
        }
      }
      {
        customer: {
          connect: { where: { node: { username: "graphlover123" } } }
        }
        products: {
          Book: {
            connect: { where: { node: { isbn: "9781492047681" } } }
          }
        }
      }
      {
        customer: {
          connect: { where: { node: { username: "graphlover123" } } }
        }
        products: {
          Book: {
            connect: [{ where: { node: { isbn: "9781617295645" } } }]
          }
          Video: { connect: { where: { node: { sku: "v001" } } } }
        }
      }
    ]
  ) {
    orders {
      orderId
      created
      customer {
        username
      }
```

```
      products {
        __typename
        ... on Book {
          title
          isbn
        }
        ... on Video {
          name
          sku
        }
      }
    }
  }
}
```

Обратите внимание, что для создания нескольких заказов в одной мутации GraphQL можно передать массив объектов:

```
{
  "data": {
    "createOrders": {
      "orders": [
        {
          "orderId": "38cfd8e4-f866-4c8a-ae97-e9e7c9e72b0b",
          "created": "2021-08-16T13:33:08.288Z",
          "customer": {
            "username": "bobbytables"
          },
          "products": [
            {
              "__typename": "Book",
              "title": "Full Stack GraphQL",
              "isbn": "9781617297038"
            }
          ]
        },
        {
          "orderId": "597ba737-de86-4772-b541-6a0bf4a25817",
          "created": "2021-08-16T13:33:08.288Z",
          "customer": {
            "username": "graphlover123"
          },
          "products": [
            {
              "__typename": "Book",
              "title": "Graph Algorithms",
              "isbn": "9781492047681"
```

```
            }
          ]
        },
        {
          "orderId": "dfc08de3-68f9-407c-8c72-1b02eb7a9b4e",
          "created": "2021-08-16T13:33:08.288Z",
          "customer": {
            "username": "graphlover123"
          },
          "products": [
            {
              "__typename": "Video",
              "name": "Intro To Neo4j 4.x",
              "sku": "v001"
            },
            {
              "__typename": "Book",
              "title": "Graph-Powered Machine Learning",
              "isbn": "9781617295645"
            }
          ]
        }
      ]
    }
  }
}
```

Теперь, создав несколько заказов и связав с ними книги и видео, рассмотрим, как можно разбить получаемые результаты на страницы.

9.2. Разбиение на страницы с помощью GraphQL

Многие приложения отображают данные в виде таблиц или списков. При заполнении этих таблиц и списков часто имеет смысл запрашивать с сервера только часть результатов, для заполнения лишь текущей видимой части списка или таблицы. Например, в контексте нашего интернет-магазина мы можем решить отображать список всех заказов, отсортированных в хронологическом порядке, или разрешить конкретному пользователю просматривать все свои заказы. Однако заказов может быть тысячи или даже миллионы, и было бы нежелательно получать сразу все эти заказы с сервера (так как это потребовало бы передать по сети слишком большой объем данных).

Было бы предпочтительнее разбить заказы на страницы и запрашивать только определенные фрагменты (или страницы) для отображения в приложении. Например, можно сначала запросить первые 20 заказов, отсортированных по дате создания. Затем, когда пользователь достигнет конца списка с первыми 20 заказами, запросить с сервера следующую страницу с результатами. GraphQL предлагает два типа разбиения на страницы: *по смещению* и с помощью *курсора*.

9.2.1. *Разбиение на страницы по смещению*

Механизм разбиения на страницы по смещению использует два аргумента, обычно называемых `limit` и `offset`. На практике часто используется третий аргумент `sort`, определяющий порядок сортировки массива. Аргумент `limit` указывает количество результатов для возврата, а `offset` – это количество объектов, которые нужно пропустить, и увеличивается на значение `limit` для выборки следующей страницы. Например, чтобы разбить результаты на страницы, содержащие по 10 записей, для получения первой страницы нужно указать в аргументе `offset` смещение 0 и в аргументе `limit` количество записей 10, для получения второй страницы в таком случае нужно будет указать смещение 10 и количество записей 10 и т. д.

Давайте представим, что в нашем приложении имеется страница «Просмотр заказов», в которой отображается таблица с заказами, отсортированными по дате создания. Запрос GraphQL для загрузки всех этих данных мог бы выглядеть примерно так, как показано в листинге 9.13.

Листинг 9.13. Запрос всех заказов, отсортированных по дате создания

```
query {
  orders(options: { sort: { created: DESC } }) {
    orderId
    created
  }
}
```

Этот запрос вернет *все* заказы. А если заказов миллионы? В таком случае серверу придется отправить слишком много данных по сети, и пользователь будет вынужден долго ждать, пока страница загрузится! В любом случае наше приложение способно отобразить только определенное количество заказов за раз, поэтому не имеет смысла запрашивать сразу все данные. Вместо этого *разобьем* результаты на страницы и вернем только ту часть, которую необходимо отобразить в приложении в данный момент. Выберем размер страницы равным 2 и запросим первую страницу, как показано в листинге 9.14.

Листинг 9.14. Запрос списка заказов с разбиением на страницы по смещению

```
query {
  orders(options: { limit: 2, offset: 0, sort: { created: DESC } }) {
    orderId
    created
  }
}
```

Определение количества страниц

Затем мы увеличим значение смещения `offset`, чтобы получить следующую страницу. Но как узнать, сколько всего страниц получится? Часто бывает жела-

тельно показать в приложении общее количество доступных страниц, чтобы пользователь знал, с каким объемом данных он имеет дело. Для этой цели можно использовать *счетные запросы*. Библиотека Neo4j GraphQL генерирует счетное поле запроса для каждого типа, в котором возвращает количество узлов этого типа в базе данных, как показано в листинге 9.15. Клиентское приложение может использовать это число для определения общего количества страниц.

Листинг 9.15. Запрос списка заказов с разбиением на страницы по смещению с включением поля ordersCount

```
query {
  ordersCount
  orders(options: { limit: 2, offset: 0, sort: { created: DESC } }) {
    orderId
    created
  }
}
```

Если используется фильтр, например для фильтрации заказов по дате создания, то можно аргумент с фильтром передать счетному полю, чтобы оно верно определило общее количество результатов, и затем определить количество страниц для отображения на стороне клиента.

9.2.2. Разбиение на страницы с помощью курсора

Разбиение на страницы с помощью курсора – еще одна модель, часто используемая на практике. В ней вместо числового смещения для разбиения результатов на страницы используется курсор – строковое значение, идентифицирующее последний объект на странице результатов. Чтобы увидеть, как действует разбиение на страницы с помощью курсора, представим, что наше приложение имеет страницу для просмотра заказов, принадлежащих конкретному пользователю, чтобы пользователь мог просмотреть все свои заказы, отсортированные по дате создания.

Чтобы разбить результаты на страницы с помощью курсора, начнем с запроса поля `ordersConnection` вместо `orders`. Поле `ordersConnection` – это так называемый объект соединения Relay. Для начала посмотрим, как используются соединения Relay, а затем исследуем их модель (листинг 9.16).

Листинг 9.16. Использование поля ordersConnection типа соединения Relay

```
query {
  users(where: { username: "graphlover123" }) {
    username
    ordersConnection(sort: { node: { created: ASC } }) {
      edges {
        node {
          created
```

```
          orderId
        }
      }
    }
  }
}
```

Обратите внимание, что список выборки для поля `ordersConnection` теперь включает вложенные поля `edges` и `node`. Давайте разберемся с ними подробнее.

Модель соединения Relay

Эти поля *соединения* генерируются библиотекой Neo4j GraphQL для каждого поля отношения и соответствуют спецификации соединений курсоров Relay (**https://relay.dev/graphql/connections.htm**), обычно называемой *спецификацией Relay*, или *соединениями Relay*. *Relay* – это клиент GraphQL, включающий множество функций. Обсуждение этих функций выходит за рамки данной книги, тем не менее отмечу, что эта спецификация Relay широко используется для реализации разбиения на страницы в GraphQL и вводит концепцию типа соединения.

Типы соединений обеспечивают два стандартных метода разбиения результатов на страницы: с помощью аргументов `first` и `after` полей и предоставляя курсоры и другую метаинформацию о наборе результатов, например доступность каких-либо других результатов для клиента, чтобы тот мог получить их, выполняя разбиение на страницы.

Согласно спецификации Relay, каждый объект соединения должен содержать поле массива `edges` и поле объекта `pageInfo`. Поле `edges` содержит список *типов ребер*, определенных спецификацией Relay, которые представляют отношения, соединяющие узлы в графе. Поле `pageInfo` содержит метаданные о странице, такие как `hasNextPage` и `hasPreviousPage`, а также курсоры, которые можно использовать для получения следующей и предыдущей страниц: `startCursor` и `endCursor`. Кроме того, библиотека Neo4j GraphQL добавляет поле `totalCount`, сообщающее общее количество ребер.

Давайте посмотрим, как используются эти механизмы на практике (листинг 9.17). Добавим аргумент `first: 2` в предыдущий запрос, чтобы разбить заказы на страницы с размером 2. Также запросим объект `pageInfo` и поле `totalCount`.

Листинг 9.17. Использование объекта pageInfo для получения метаданных

```
query {
  users(where: { username: "graphlover123" }) {
    username
    ordersConnection(first: 2, sort: { node: { created: ASC } }) {
      totalCount
      pageInfo {
        endCursor
        hasNextPage
        hasPreviousPage
```

```
      }
      edges {
        node {
          created
          orderId
        }
      }
    }
  }
}
```

Теперь в набор результатов будут включены первые два заказа, заключенные в массив `edges`, а также объект метаданных `pageInfo`, содержащий курсор `endCursor`, который можно использовать для получения следующей страницы с результатами:

```
{
  "data": {
    "users": [
      {
        "username": "graphlover123",
        "ordersConnection": {
          "totalCount": 3,
          "pageInfo": {
            "endCursor": "YXJyYXljb25uZWN0aW9uOjE=",
            "hasNextPage": true,
            "hasPreviousPage": false
          },
          "edges": [
            {
              "node": {
                "created": "2021-08-15T13:43:15.117Z",
                "orderId": "dfdebf08-3ce5-494e-9843-d5286f4dc8f4"
              }
            },
            {
              "node": {
                "created": "2021-08-16T13:33:08.288Z",
                "orderId": "dfc08de3-68f9-407c-8c72-1b02eb7a9b4e"
              }
            }
          ]
        }
      }
```

```
    ]
  }
}
```

Чтобы запросить следующую страницу, включим значение `endCursor` в виде аргумента `after` поля `ordersConnection` (листинг 9.18).

Листинг 9.18. Использование курсора для получения следующей страницы с заказами

```
query {
  users(where: { username: "graphlover123" }) {
    username
    ordersConnection(
      first: 2
      after: "YXJyYXljb25uZWN0aW9uOjE="
      sort: { node: { created: ASC } }
    ) {
      totalCount
      pageInfo {
        endCursor
        hasNextPage
        hasPreviousPage
      }
      edges {
        node {
          created
          orderId
        }
      }
    }
  }
}
```

На этот раз мы получим `hasNextPage` со значением `false`, сообщающим, что клиент не сможет получить дополнительные результаты:

```
{
  "data": {
    "users": [
      {
        "username": "graphlover123",
        "ordersConnection": {
          "totalCount": 3,
          "pageInfo": {
            "endCursor": "YXJyYXljb25uZWN0aW9uOjI=",
```

```
            "hasNextPage": false,
            "hasPreviousPage": true
          },
          "edges": [
            {
              "node": {
                "created": "2021-08-16T13:33:08.288Z",
                "orderId": "597ba737-de86-4772-b541-6a0bf4a25817"
              }
            }
          ]
        }
      }
    ]
  }
}
```

Модель соединений Relay – удобный и стандартный способ разбиения на страницы. Типы ребер, определяемые спецификацией Relay, обеспечивают еще одну мощную особенность графовой модели данных, с которой мы познакомимся далее: свойства отношений.

9.3. Свойства отношений

В графовой модели данных *свойства отношений* – это атрибуты, хранящиеся в отношениях, которые используются для представления значений, имеющих смысл в контексте узлов, связанных отношением. Например, как в нашей модели данных магазина можно представить количество товара в заказе? Лучший способ представить это понятие количества – сохранить свойство в отношении `CONTAINS`, которое представляет количество этого товара (книги или видео) в заказе.

Рис. 9.3. Модель данных интернет-магазина, дополненная свойствами отношений

На рис. 9.3 показана дополненная модель данных, в которой мы добавили целочисленное свойство `quantity` в отношение `CONTAINS`. Теперь, если клиент пожелает разместить заказ на приобретение двух экземпляров книги «Приложения GraphQL полного цикла», мы сможем присвоить этому свойству значение 2. Но как представить эти свойства в GraphQL API?

9.3.1. *Интерфейсы и директива @relationship*

Мы использовали директиву `@relationship` с библиотекой Neo4j GraphQL, чтобы указать тип и направление отношения в графе, используя аргументы `type` и `direction`. Директива `@relationship` также принимает необязательный аргумент `properties`, с помощью которого можно задавать свойства отношений. Аргумент `properties` принимает имя типа интерфейса, определяющего поля GraphQL для отображения в свойства отношений.

Чтобы представить поля для свойств отношений, сначала нужно определить тип интерфейса, включающий эти поля. Поскольку мы добавляем только одно поле `quantity` в отношение `CONTAINS`, создадим интерфейс `Contains` с одним полем. Затем в директиве `@relationship`, используемой в поле `Order.products`, добавим `properties: "Contains"`, чтобы указать, что используем интерфейс `Contains` для представления свойств отношения `CONTAINS`. Дополненные определения типов GraphQL показаны в листинге 9.19; давайте продолжим и обновим определения в index.js.

Листинг 9.19. Использование интерфейса для представления свойств отношений в GraphQL

```
interface Contains {
  quantity: Int
}

type User {
  username: String
  orders: [Order!]! @relationship(type: "PLACED", direction: OUT)
}

type Order {
  orderId: ID! @id
  created: DateTime! @timestamp(operations: [CREATE])
  customer: User! @relationship(type: "PLACED", direction: IN)
  products: [Product!]!
  @relationship(type: "CONTAINS", direction: OUT, properties: "Contains")
}

type Video {
  title: String
  sku: String
}

type Book {
```

```
  title: String
  isbn: String
}

union Product = Video | Book
```

Обратите внимание: поскольку `Product` является типом объединения, представляющим типы `Video` и `Book`, мы фактически добавили определение свойства отношения `quantity` для обоих этих типов – отличный пример возможностей абстрактных типов! После обновления определений типов в index.js нужно перезапустить приложение, чтобы изменения вступили в силу.

9.3.2. Создание свойств отношений

Теперь, добавив в определения типов GraphQL свойство отношения `quantity`, посмотрим, как можно использовать его. Сначала создадим новый заказ, но на этот раз мы разместим заказ на 10 экземпляров книги «Full Stack GraphQL». Для этого включим `edge: {quantity: 10}` в объект `connect` для входного объекта в мутации `createOrders` (листинг 9.20).

Листинг 9.20. Использование свойства отношения в мутации GraphQL

```
mutation {
  createOrders(
    input: {
      customer: {
        connect: { where: { node: { username: "graphlover123" } } }
      }
      products: {
        Book: {
          connect: {
            edge: { quantity: 10 }
            where: { node: { title: "Full Stack GraphQL" } }
          }
        }
      }
    }
  ) {
    orders {
      created
      orderId
      productsConnection {
        edges {
          quantity
          node {
            ... on Book {
```

```
            title
          }
        }
      }
    }
  }
}
```

Теперь у нас есть поле `quantity` в объектах ребер в поле `productsConnection`, которое сообщает, что этот заказ содержит 10 экземпляров книги, как видно из результатов запроса:

```
{
  "data": {
    "createOrders": {
      "orders": [
        {
          "created": "2021-08-18T22:17:28.285Z",
          "orderId": "48faa3f4-553b-42ed-a08f-e7781aed3c17",
          "productsConnection": {
            "edges": [
              {
                "quantity": 10,
                "node": {
                  "title": "Full Stack GraphQL"
                }
              }
            ]
          }
        }
      ]
    }
  }
}
```

Благодаря спецификации соединений Relay мы теперь можем представлять и использовать свойства отношений в GraphQL!

9.4. В заключение

Теперь вы знаете, как использовать GraphQL, графовые базы данных, React и облачные службы для создания и защиты веб-приложений полного цикла. Основная цель этой книги заключалась в том, чтобы показать, как разные части стека GraphQL сочетаются друг с другом. Давайте вспомним, о чем рассказывалось в нашей книге, и наметим некоторые пути для дальнейшего изучения.

В первой части мы познакомились с графовым мышлением, GraphQL и графовой базой данных Neo4j. Мы узнали о преимуществах GraphQL, о том, как писать запросы GraphQL, и об основных подходах к созданию серверов GraphQL. Мы расширили наше графовое мышление и охватили графовые базы данных, познакомившись с Neo4j и языком запросов Cypher. Во второй части мы исследовали возможности фреймворка React JavaScript, предназначенного для создания пользовательских интерфейсов, и использовали Apollo Client для получения данных из GraphQL API. Наконец, в третьей части мы рассмотрели способы аутентификации и авторизации в приложениях GraphQL API и React, а также развертывание приложений в управляемых облачных службах, таких как Auth0, Neo4j AuraDB, Netlify и AWS Lambda.

Библиотека Neo4j GraphQL является основным компонентом приложений GraphQL полного цикла и помогает создавать мощные GraphQL API с поддержкой Neo4j, без написания шаблонного кода; однако библиотека обладает множеством других возможностей, которые не были охвачены в этой книге. Когда вы начнете создавать свои приложения с помощью GraphQL, я советую поближе познакомиться с этими возможностями, такими как работа с агрегатами, и с другими директивами схемы, такими как `@cypher` и `@auth`, которые позволят вам обогатить ваши GraphQL API. Лучшим ресурсом для дальнейшего изучения библиотеки Neo4j GraphQL является документация, доступная по адресу **https://neo4j.com/docs/graphql-manual/current/**.

Еще одна тема, которую я хотел бы включить в книгу, – работа с фреймворками и инструментами React, упрощающими создание пользовательских интерфейсов. Next.js – один из таких фреймворков. Он основан на React и предлагает множество дополнительных возможностей, отсутствующих в React. Благодаря Routes API фреймворк Next.js позволяет даже создавать GraphQL API – весьма интересный подход к размещению серверной логики. Отличным практическим введением в Next.js может служить учебное пособие, включенное в документацию Next.js: **https://nextjs.org/docs/getting-started**.

Продолжить знакомство с графовыми базами данных и Neo4j можно с помощью бесплатных онлайн-курсов в Neo4j GraphAcademy. Они охватывают самые разные темы, включая те, которые не рассматриваются в этой книге, такие как наука о графовых данных и создание приложений с использованием разных языков и фреймворков. Список курсов вы найдете на сайте GraphAcademy: **https://graphacademy.neo4j.com/**.

Наконец, у меня есть свой блог, где я подробно освещаю многие из этих тем: **https://lyonwj.com/**.

9.5. Упражнения

1. Цена товаров в интернет-магазине может меняться, например снижаться в период акции. Добавьте свойство отношения для хранения цены каждого товара, оплаченного в заказе.
2. Напишите поле директивы `@cypher` для вычисления промежуточной суммы заказа. Обязательно учитывайте количество каждого товара, включенного в заказ.

3. Напишите запрос GraphQL для разбиения на страницы списка товаров, включенных в заказ, сначала с использованием смещения, а затем с помощью курсора. Проверьте также, сможете ли вы с помощью курсора перейти с последней страницы на первую.

Итоги

- GraphQL поддерживает два абстрактных типа, которые можно использовать для представления нескольких конкретных типов: объединения и интерфейсы.
- Интерфейсы используются, когда конкретные типы имеют общие поля, и их можно рассматривать как контракт, определяющий требования для реализации интерфейса.
- Объединения не разделяют идею контракта и могут использоваться, когда конкретные типы не имеют общих полей.
- Два распространенных подхода, используемых в GraphQL для разбиения результатов на страницы, включают использование смещений и курсоров. Разбиение на страницы по смещению основано на использовании числовых смещений результатов, а разбиение на страницы с помощью курсора – на использовании строк-курсоров, формируемых сервером и не имеющих какого-то особого смыслового значения.
- Спецификация Relay определяет общий тип соединения, который можно использовать для разбиения на страницы на основе курсора.
- Типы соединений Relay также можно использовать для моделирования свойств отношений.

Предметный указатель

Символы

$first параметр 94
@auth деректива схемы 163
  правила и операции 163
@auth директива схемы 158, 162
  allow правило 168
  bind правило 171
  isAuthenticated правило 164
  roles правило 165, 173
  where правило 170
@client директива 149
@cypher директива 104
  вычисляемые поля с объектами и массивами 106
  вычисляемые скалярные поля 105
  настраиваемые поля запроса верхнего уровня 107
@cypher директива схемы 162
@id директива 232
@ignore директива 108
@neo4j/graphql пакет 87
@neo4j/introspector пакет 110
@relationship директива схемы GraphQL 89, 231, 249
@timestamp директива 232
-A флаг 205
__typename метаполе 229
:use neo4j команда 194
--use-npm флаг 117

А

абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) 132
абстрактные типы 228
  интерфейсы 228
  использование в мутациях GraphQL 234
  использование с библиотекой Neo4j GraphQL 230
    создание сервера GraphQL 233
    моделирование API интернет-магазина 231
  объединения 229
авторизация и аутентификация
  @auth директива схемы 162
    allow правило 168
    bind правило 171
    isAuthenticated правило 164
    roles правило 165, 173
    where правило 170
  JSON Web Token 158
агрегаты 79

Б

безиндексная смежность 70
бессерверные функции 216
  AWS Lambda
    GraphQL как бессерверная функция 216
    добавление собственного домена в Netlify 222
    преобразование GraphQL API в функцию Netlify 219

В

вложенные запросы 94
вложенные мутации 145
встроенные фрагменты 229

Г

генерирование схемы из определений типов 88
графовая модель свойств 33, 34, 66
  метки узлов 66
  отношения 67
  свойства 68
графы 46
  Neo4j
    графовая модель свойств 66
    моделирование графовых данных 65
    обзор 64
  в GraphQL 46
  вопросы моделирования данных 69
    выбор направления отношений 70
    индексы 70
    специфика отношений 70

узел и отношение 70
узел и свойство 69
данные приложения 44
добавление в API постраничного просмотра и упорядочения 50
корневые функции разрешения 59
моделирование API с применением определений типов 46
объединение определений типов и функций разрешения 57
переменные 138
реализация функций разрешения 59
сигнатура функций разрешения 53
функции разрешения 53
функции разрешения по умолчанию 54

И

избыточная выборка 25
интеграция с базой данных 83
интерфейс командной строки (Command Line Interface, CLI) 188
интроспекция 27

К

кеширование 27
компоненты 115
иерархия 117

Л

локальные поля 147

М

модель на доске 66
мутации 143
вложенные 145
изменение и удаление данных 146
создание отношений 145
создание узлов 143

Н

недостаточная выборка 25

О

объединения 229
определение схемы GraphQL в существующей базе данных 110
определения типов
моделирование API 46
отношения 72

П

проблема *n* + 1 запросов 82
продуктивность разработчика 188
пространственные данные 102
псевдонимы 73

Р

разбиение на страницы 242
добавление в API 49
определение количества страниц 244
по смещению 243
с помощью курсора 244
развертывание 188
недостатки подхода 189
обзор подхода 189
преимущества подхода 188
развертывание с помощью Netlify Build 201
добавление сайта в Netlify 202
настройка переменных окружения для сборок Netlify 210
предварительное развертывание в Netlify 213
разработка на основе GraphQL 46
реактивные переменные 147

С

свойства 72
свойства отношений 248
сеть доставки контента (Content Delivery Network, CDN) 190
сопоставление с шаблоном 34
состояние и подключаемые обработчики React Hooks 124

Т

типовой код 83
типы даты и времени 100
типы интерфейсов 228
типы ребер 245

У

узлы 72
метки узлов 66

упорядочение
добавление в API 49
упорядочение и разбиение на страницы 93
управление доступом на основе ролей (Role-Based Access Control, RBAC) 173
управление состоянием клиента 147
управляемые службы 188

Ф

фильтрация 95
where аргумент 95
вложенные фильтры 96
выборки 98
логические операторы 97
по расстояниям 103
фильтры по полям с типами Date DateTime 101
фрагменты GraphQL 139
функции разрешения 24
корневые 59
массивов 60
объектные 61
по умолчанию 54
реализация 59
реализация собственных функций разрешения 108
сигнатура 54
скалярные 61
функция как услуга (Function as a Service, Faa) 188

Э

элементы 115

A

admin роль 167
after аргумент поля 245
allBusiness функция разрешения 56
allow правило 163, 168
ALTER CURRENT USER команда Cypher 193
AND оператор 97
Apollo Client 33, 131
внедрение в иерархию компонентов 133
выполнение запросов 132
добавление в приложение React 131
использование переменных 140
кеширование 141
обновление кешированных результатов 141
обработка ответов 41
обработчики 134
опрос 141
повторная выборка 141
создание экземпляра 132
установка 131
ApolloProvider компонент 134, 177
Apollo Server 33, 57
объединение определений типов и функций разрешения 57
серверная часть GraphQL 39
apollo-server пакет 87
Apollo Studio 29, 58
выполнение запросов с помощью 62
App компонент 121, 134, 177
args аргумент 53
aud утверждение 158
AuraDB Free тип базы данных 191
Auth0Provider компонент 177
auth0-react библиотека 175, 180
Auth0 служба 172
и React 175
настройка 172
averageStars поле 164, 183
AWS Lambda 216
GraphQL как бессерверная функция 216
добавление собственного домена в Netlify 222
клиент командной строки Netlify dev 218
преобразование GraphQL API в функцию Netlify 219
AWS Lambda управляемая служба 190

B

bind правило 163, 171
Business.avgStars функция разрешения 61
businessBySearchTerm функция разрешения 59
BUSINESS_DETAILS_FRAGMENT переменная 140
businessDetails фрагмент 140
businessId поле 146
BusinessResults компонент 124, 136, 184
Business.reviews функция разрешения 60
Business метка узла 86
Business тип 54, 105, 106, 140
Business узел 68

C

CartesianPoint пространственный тип 102

Category метка узла 86
CDN (Content Delivery Network сеть доставки контента) 190
CLI (Command Line Interface интерфейс командной строки) 188
connect аргумент 145
connect объект 250
CONTAINS отношение 249
context аргумент 53
context объект 54
created поле 237
createOrders мутация 237, 250
Create React App 32
create-react-app инструмент 205
createUsers мутация GraphQL 234
create аргумент 145
CREATE команда 73
Customer объект 228
Customer тип 228
Cypher 34, 72
  CREATE команда 73
  MATCH команда 78
  MERGE команда 76
  SET команда 73
  WHERE оператор 78
  агрегаты 79
  определение ограничений 77
    ограничение ключа узла 78
    ограничение существования свойства 78
    ограничение уникальности 78
  сопоставление с образцом 72
  сопоставление с шаблоном 34

D

data объект 136
DateTime тип 100
Date тип 100
db объект 54
DEBUG переменная окружения 91
deleteBusinesses мутация 147
direction аргумент 89, 249
DOM модель 30

E

edges поле 245
edges поле-массив 245
Employee объект 228
exp утверждение 158
extend ключевое слово GraphQL 166

F

FaaS (Function as a Service) функция как услуга 188
firstName поле 228
first аргумент 49, 94
first аргумент поля 245

G

getAccessTokenSilently функция 180
GET запрос 25, 57
git add команда 205
git commit команда 206
git push команда 207
git status команда 203, 213
GraphiQL 28
GraphQL
  недостатки
    ограничения 28
  Apollo Server 57
  Neo4j Aura 189
    выгрузка данных в 196
    исследование графа с помощью Neo4j Bloom 198
    создание кластера 191
  абстрактные типы 228
    интерфейсы 228
    использование в мутациях GraphQL 234
    использование с библиотекой Neo4j GraphQL 230
    объединения 229
  вложенные запросы 94
  генерирование схемы из определений типов 88
  графы
    добавление в API постраничного просмотра и упорядочения 50
    корневые функции разрешения 59
    объединение определений типов и функций разрешения 57
    реализация функций разрешения 59
    сигнатура функций разрешения 53
    функции разрешения 53
    функции разрешения по умолчанию 54
  добавление своей логики 104
    @cypher директива 104

вычисляемые поля с объектами и массивами 106
вычисляемые скалярные поля 105
настраиваемые поля запроса верхнего уровня 107
реализация собственных функций разрешения 108
запросы 22
инструменты 28
Apollo Studio 29
GraphiQL 28
GraphQL Playground 29
интеграция с базой данных 83
мутации 143
вложенные 145
изменение и удаление данных 146
создание отношений 145
создание узлов 143
настройка проекта 84
Neo4j 84
недостатки 27
кеширование 27
обзор 18, 20
определение схемы GraphQL в существующей базе данных 110
определения типов 20
основы запросов 90
преимущества 25
графы 26
избыточная и недостаточная выборка 25
интроспекция 27
спецификация 26
пространственные данные 102
разбиение на страницы 242
определение количества страниц 244
по смещению 243
с помощью курсора 244
развертывание с помощью Netlify Build 201
добавление сайта в Netlify 202
настройка переменных окружения для сборок Netlify 210
предварительное развертывание в Netlify 213
распространенные проблемы 82
низкая производительность 82
проблема *n* + 1 запросов 82
продуктивность разработчиков 83
типовой код 83
упорядочение и разбиение на страницы 93
управление состоянием клиента 147
фильтрация 95
where аргумент 95
вложенные фильтры 96
выборки 98
логические операторы 97
по расстояниям 103
фильтры по полям с типами Date и DateTime 101
фрагменты 139
GraphQL Playground 29
graphql-plugin-auth пакет 162
GraphQLResolveInfo объект 53
graphql пакет 87

H

hireDate поле 228
Hooks класс 116

I

ID поле 48
IN_CATEGORY тип отношения 86
info аргумент 53
InMemoryCache кеш 132
isAuthenticated переменная 179
isAuthenticated правило 163, 164
isStarred поле 148
iss утверждение 158

J

jsonwebtoken пакет 159
JSX 31, 116
JWKS (JSON Web Key Set) 163
JWT (JSON Web Token) 158
JWT_SECRET переменная окружения 162

L

lastName поле 228
limit аргумент поля 243
LIMIT оператор 94
LocalDateTime тип 100

M

MATCH инструкция 35
MATCH команда 78
MATCH оператор 75

MERGE команда 76
moviesByTitle функция разрешения 40
MovieSearch компонент React 38
Mutation тип 48, 84
mutation тип операции 53

N

name поле 54
Neo4j
  вопросы моделирования данных 69
    выбор направления отношений 70
    индексы 70
    специфика отношений 70
  инструменты
    Neo4j Browser 71
    Neo4j Desktop 71
  клиентские драйверы 79
  моделирование графовых данных 65
    графовая модель свойств 66
    ограничения базы данных и индексы 69
  обзор 64
Neo4j Aura 189
  выгрузка данных в 196
  исследование графа с помощью Neo4j Bloom 198
  подключение к кластеру 193
  создание кластера 191
Neo4j AuraDB 35
Neo4j Bloom 198
Neo4j Browser 35, 71
Neo4j Desktop 35, 71
neo4j-driver пакет 87
Neo4j GraphQL библиотека 37
  вложенные запросы 94
  генерирование схемы из определений типов 88
  добавление своей логики 104
    @cypher директива 104
    вычисляемые поля с объектами и массивами 106
    вычисляемые скалярные поля 105
    настраиваемые поля запроса верхнего уровня 107
    реализация собственных функций разрешения 108
  интеграция с базой данных 83
  использование с абстрактными типами 230
    создание сервера GraphQL 233
    использование в мутациях GraphQL 234
    моделирование API интернет-магазина 231
  настройка проекта 84
    Neo4j 84
  определение схемы GraphQL в существующей базе данных 110
  основы запросов 90
  пространственные данные 102
  распространенные проблемы 82
    низкая производительность 82
    проблема *n* + 1 запросов 82
    продуктивность разработчиков 83
    типовой код 83
  упорядочение и разбиение на страницы 93
  фильтрация 95
    where аргумент 95
    вложенные фильтры 96
    выборки 98
    логические операторы 97
    по расстояниям 103
    фильтры по полям с типами Date и DateTime 101
Neo4jGraphQL конструктор 109, 111
Neo4j база данных
  Cypher язык запросов 34
  графовая модель свойств 34
  инструменты 35
    Neo4j AuraDB 35
    Neo4j Browser 35
    Neo4j Desktop 35
    Neo4j GraphQL 37
    драйвер Neo4j JavaScript 37
    клиентские драйверы 37
neo4j пользователь базы данных 194
Netlify Build, развертывание с помощью 201
  добавление сайта в Netlify 202
  настройка переменных окружения для сборок Netlify 210
  предварительное развертывание в Netlify 213
Netlify dev клиент командной строки 218
node поле 245
npm run build команда 208
npm start команда 120

O

obj аргумент 53

offset аргумент 50
offset аргумент поля 243
orderBy аргумент 94
ORDER BY оператор 94
orderId поле 237
Order.products поле 232
ordersConnection поле 245, 247
OR оператор 97

P

pageInfo поле 245
Person интерфейс 228
play grandstack команда 196
plugins объект 162
Point пространственный тип 102
POST запрос 57
Promise объект 54
properties аргумент поля 249
push-to-cloud команда 197

Q

quantity свойство отношения 250
quantity целочисленное поле 249
Query.people функция разрешения 156
Query тип 48, 84
query тип операции 53

R

RBAC (Role-Based Access Control управление доступом на основе ролей) 173
React 30
  Apollo Client 131
    jghjc 141
    внедрение в иерархию компонентов 133
    выполнение запросов 38, 132
    добавление в приложение React 131
    использование переменных 140
    кеширование 141
    обновление кешированных результатов 141
    обработка ответов 41
    обработчики 134
    повторная выборка 141
    создание экземпляра 132
    установка 131
  Apollo Server
    серверная часть GraphQL 39
  GraphQL
    мутации 143
    переменные 138
    создание отношений 145
    управление состоянием клиента 147
    фрагменты 139
  GraphQL мутации
    вложенные 145
    изменение и удаление данных 146
  JSX 31
  React App инструмент 117
  React Hooks 124
  react-scripts пакет 119
  библиотеки компонентов 31
  инструменты 32
    Create React App 32
    React Chrome DevTools 32
  компоненты 31
  обзор 115
    иерархия компонентов 117
    и элементы JSX 116
    компоненты 116
  состояние и подключаемые обработчики React Hooks 124
REACT_APP_GRAPHQL_URI переменная окружения 212
React App инструмент 117
React Chrome DevTools 32
React.createElement() функция 116
React Hooks 124
react-scripts инструмент 218
react-scripts пакет 119
recommended поле 106
refetch функция 142
Relay модель соединения 245
RETURN оператор 73
REVIEWS тип отношения 86
Review.user функция разрешения 61
Review метка узла 86
roles правило 163, 165, 173
rules аргумент 168

S

setSelectedCategory функция 127
setState функция 124
SET команда 73
shippingAddress поле 228
sort аргумент поля 243

starredVar реактивная переменная 150
String значение 48
subscription тип операции 53
sub утверждение 158

T

title аргумент 40
totalCount поле 245
type аргумент 89, 249

U

updateBusinesses мутация 146
updateUsers мутация 237
update аргумент 146
useAuth0 обработчик 179
useQuery обработчик 131, 134, 142
userId поле 48
userId свойство узла 170
username поле 228
User метка узла 86
User объект 48
User тип 68, 166
User узел 68
useState обработчик 127

W

waitTime значение 109
web-react каталог 131, 175
where аргумент 95, 138
WHERE оператор 78
where правило 163, 170
WROTE тип отношения 86

Y

yarn диспетчер пакетов 117

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине:
www.galaktika-dmk.com

Уильям Лион

**Разработка веб-приложения GraphQL
с React, Node.js и Neo4j**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Печать цифровая.
Усл. печ. л. 21,29. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Язык запросов API GraphQL заметно упрощает обмен данными с серверами, позволяя приложениям получать данные в виде простых для понимания графов. Преимущества GraphQL можно усилить за счет таких графовых инструментов и хранилищ данных, как React, Apollo и Neo4j. Подход к разработке графовых приложений полного цикла обеспечивает согласованную модель данных от начала до конца, повышая продуктивность разработчиков.

Эта книга научит вас создавать графовые веб-приложения с использованием GraphQL, React, Apollo и базы данных Neo4j, которые все вместе называются GRANDstack. На практических примерах вы увидите, как элементы GRANDstack сочетаются друг с другом. Попутно создадите и развернете в облаке полноценное веб-приложение, поддерживающее поиск, аутентификацию и многое другое. А также узнаете, как развертывать комплексные приложения, в полной мере использующие производительность GraphQL.

«Для всех, кто желает создавать потрясающие приложения с помощью GraphQL».
Густаво Гомес, Troido

«Эта книга превзошла все мои ожидания. Содержит простые примеры и представляет все, что может понадобиться для создания сложного и современного приложения».
Данило Зекович, NeonCat.io

«Хорошо продуманное и написанное руководство по созданию современных приложений GraphQL».
Хосе Сан Леандро, ioBuilders

«Отличный источник знаний, необходимых для разработки приложений GraphQL с использованием передовых технологий, таких как Neo4j и React».
Теофанис Деспудис, WP Engine

Для веб-разработчиков полного цикла.
Опыт работы с GraphQL или графовыми базами данных не требуется.

Рассматриваемые темы:

- создание серверной части GraphQL с использованием Neo4j;
- аутентификация и авторизация с помощью GraphQL;
- разбиение на страницы и абстрактные типы GraphQL;
- разработка пользовательского интерфейса с использованием React и Apollo Client;
- развертывание в облаке с помощью Netlify, AWS Lambda, Auth0 и Neo4j Aura.

Уильям Лион — консультант в Neo4j, где он помогает разработчикам, и автор блога lyonwj.com.

ISBN 978-5-93700-185-6

9 785937 001856 >

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК Издательство
www.дмк.рф