O'REILLY®

Разработка на JavaScript

Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron

Адам Д. Скотт

JavaScript Everywhere

Building Cross-Platform Applications with GraphQL, React, React Native, and Electron

Adam D. Scott

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Разработка на JavaScript

Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron

Адам Д. Скотт

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

Адам Д. Скотт

Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron

Серия «Бестселлеры O'Reilly»

Перевел с английского *Д. И. Акуратер*

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*А. Юринова*
Литературный редактор	*Ю. Зорина*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Г. Шкатова*
Верстка	*Л. Егорова*

ББК 32.988.02-018
УДК 004.738.5

Скотт Адам Д.

С44 Разработка на JavaScript. Построение кроссплатформенных приложений с помощью GraphQL, React, React Native и Electron. — СПб.: Питер, 2021. — 320 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1462-7

Что такое современный JavaScript? Когда-то он просто добавлял интерактивности к окнам веб-браузера, а теперь превратился в основательный фундамент мощного и надежного софта. Разработчики любого уровня смогут использовать JavaScript для создания API, веб-, мобильных и десктопных приложений.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1492046981 англ.

Authorized Russian translation of the English edition of JavaScript Everywhere ISBN 9781492046981 © 2020 Adam D. Scott.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1462-7

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 07.2021. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 29.06.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 700. Заказ 0000.

Краткое содержание

Оглавление

«Разработка на JavaScript» — это невероятная книга, которая даст все, что вам нужно, для создания приложений с помощью JavaScript на любой платформе. Сейчас JavaScript действительно встречается буквально везде, и в этой книге был проделан уникальный труд по совмещению информации для разработчиков всех уровней. Прочтите ее, напишите код и уверенно принимайте технологические решения.

Ева Порцелло (Eve Porcello),
разработчик ПО и инструктор в Moon Highway

«Разработка на JavaScript» — это идеальный спутник для путешествия по непрестанно меняющейся экосистеме современного JavaScript. В этой книге Адам простым и понятным способом учит использовать React, React Native и GraphQL для разработки надежных веб-, мобильных и десктопных приложений.

Пегги Райзис (Peggy Rayzis),
главный инженер в Apollo GraphQL

Посвящается моему отцу, который собрал со мной мой первый компьютер и вычитал каждую мою статью. Меня бы не было здесь без тебя. Мне тебя не хватает.

Предисловие

В 1997-м мне оставалось два года до выпуска из школы. Однажды мы с другом возились с подключенным к интернету компьютером в нашей школьной библиотеке, и он показал мне, что можно щелкнуть View -> Source и просмотреть основной код веб-страницы. Несколько дней спустя уже другой товарищ показал мне, как публиковать собственный HTML. Мой мозг просто взорвался.

С этого момента я подсел. Я начал собирать крупицы информации с сайтов, которые мне нравились, постепенно создавая из них собственный Франкен-сайт. Большую часть свободного времени я возился за собранным по частям компьютером, стоявшим у нас в столовой. Я даже написал (честно говоря, скопировал и вставил) свой первый JS-код, чтобы реализовать стили наведения на ссылки, что на тот момент еще нельзя было сделать с помощью простого CSS.

И вследствие поворота событий, который я рассматриваю как благоприятную версию фильма «Почти знаменит», мой доморощенный музыкальный сайт стал весьма популярен. Благодаря этому я начал получать по почте промо-CD и стал званным гостем на различных концертах. И все же самым важным для меня было то, что я мог общаться с людьми по всему миру. Я был скучающим подростком из глубинки, который любил музыку и при этом мог общаться с людьми, которых никогда не встретит в живую. Это ощущение как тогда, так и сейчас очень вдохновляет.

Сегодня мы можем создавать мощные приложения с помощью одних только веб-технологий, но мысль о том, чтобы приступить к этому, может пугать. API являются невидимым фоном, предоставляющим данные, View -> Source показывает конкатенированный и минифицированный код, аутентификация и безопасность таинственны, а совмещение всего этого воедино может показаться чем-то и вовсе невыполнимым. Если же взглянуть на то, что кроется за всей этой неразберихой, то можно заметить, что с помощью тех же технологий, которые я использовал еще 20 лет назад, сегодня можно не только создавать мощные веб-, десктопные или нативные мобильные приложения, но также проектировать 3D-анимацию и даже программировать роботов.

Занимаясь преподаванием, я обнаружил, что многие из нас лучше всего учатся, создавая нечто новое, разбирая это и адаптируя к конкретным случаям использования. Эту же цель преследует и моя книга. Если вы немного знакомы с HTML, CSS и JavaScript, но навыков объединить все это для создания

надежных придуманных вами приложений у вас пока маловато, то эта книга для вас. Я познакомлю вас с созданием API, который может применяться для пользовательских интерфейсов веб-, десктопных и нативных мобильных приложений. Но важнее всего то, что вы получите понимание принципов совместимости всех этих частей, которое позволит вам разрабатывать и создавать прекрасные программы.

Мне не терпится увидеть, что у вас получится.

Адам

Введение

Идея этой книги возникла у меня после написания первого десктопного приложения на Electron. Сделав карьеру веб-разработчика, я был захвачен возможностями использования веб-технологий для создания кроссплатформенных приложений. В то же время происходил рост популярности React, React Native и GraphQL. Я искал ресурсы, которые бы помогли мне научиться совмещать все эти компоненты, но ничего не нашел. Так что эта книга представляет собой руководство, которого мне тогда очень не хватало.

Ее конечная цель — научить вас использовать возможности одного только JavaScript для создания всех видов приложений.

Для кого эта книга

Она предназначена для разработчиков среднего уровня, имеющих некоторый опыт работы с HTML, CSS и JS, а также для амбициозных новичков, желающих изучить инструменты, необходимые для запуска проектов для бизнеса и любых других целей.

Структура

Структура книги подразумевает поэтапную разработку примера приложения для различных платформ. По сути, она делится на следующие разделы:

- глава 1 знакомит с настройкой среды разработки JavaScript;
- главы 2–10 посвящены созданию API с использованием Node, Express, MongoDB и Apollo Server;
- в главах 11–25 рассматриваются детали создания кроссплатформенных пользовательских интерфейсов с помощью React, Apollo и ряда других инструментов. А именно:
    - глава 11 знакомит с принципами разработки пользовательского интерфейса и рассказывает о React;

- главы 12–17 демонстрируют создание веб-приложения при помощи React, Apollo Client и CSS-in-JS;
- главы 18–20 поэтапно поясняют создание простых приложений на Electron;
- главы 21–25 учат использовать React Native и Expo для разработки мобильных приложений под iOS и Android.

Условные обозначения

В данной книге используются следующие типографские обозначения:

Курсив

Обозначает новые термины.

Интерфейс

Обозначает URL, адреса электронной почты, названия файлов и их расширения.

`Моноширинный шрифт`

Используется для примеров программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, переменные среды, операторы и ключевые слова.

`Моноширинный курсив`

Текст, который следует заменить значениями, введенными пользователем, или значениями, определяемыми контекстом.

Этот элемент означает подсказку или предложение.

Этот элемент обозначает общее примечание.

Этот элемент указывает на предупреждение или предостережение.

Использование примеров кода

Сопроводительный материал (примеры кода, упражнения и т. д.) доступен для загрузки по ссылке https://github.com/javascripteverywhere.

Цель этой книги — помочь вам справиться с задачей. В целом предложенные в ней примеры кода вы можете без проблем использовать в собственных программах и документации. Для этого не нужно спрашивать у нас разрешения, если только вы не соберетесь использовать существенную его часть. Например, написание программы, использующей несколько кусков кода из этой книги, не требует разрешения, а вот для продажи или распространения примеров из книг издательства O'Reilly разрешение уже требуется. Ответы на вопросы, использующие цитирование книги или кода, не требуют разрешения. Для включения же существенного количества примеров кода в документацию вашего продукта разрешение уже нужно.

Благодарности

Благодарю всех замечательных сотрудников O'Reilly, как бывших, так и нынешних, которые на протяжении нескольких лет были открыты и шли навстречу моим идеям. Отдельно хочу поблагодарить своего редактора Анджелу Руфино (Angela Rufino), которая давала отзывы, приободряла и делала много полезных напоминаний. Хочу также сказать спасибо Майку Лукидесу (Mike Loukides), который поддерживал меня не только кофеином, но и дружеской беседой. Выражаю признательность Дженнифер Поллок (Jennifer Pollock) за ее поддержку и воодушевление.

Я также бесконечно благодарен сообществу open source за предоставленные мне знания, которые я смог с успехом применить. Без отдельных людей и организаций, разработавших и обслуживающих множество упоминаемых мной библиотек, написание этой книги стало бы невозможным.

Научные редакторы помогли мне улучшить ее, обеспечив точность использования всех элементов. Спасибо вам, Энди Нгом (Andy Ngom), Брайан Слеттен (Brian Sletten), Максимилиано Фиртман (Maximiliano Firtman) и Зишан Чавдхари (Zeeshan Chawdhary). Вы сделали грандиозную работу по проверке кода, за что я искренне благодарен. Отдельное спасибо отправляется в адрес моего давнего коллеги и друга Джимми Уилсона (Jimmy Wilson), которому я позвонил уже в одиннадцатом часу, чтобы он просмотрел весь материал и дал отзыв. Это была существенная просьба, но он по своему обыкновению отреагировал на нее с энтузиазмом. Должен отметить, что без его помощи эта книга не появилась бы в ее нынешнем виде.

На протяжении всей взрослой жизни мне невероятно везло с умными, увлеченными и поддерживающими меня коллегами. Я получил много технических

и жизненных уроков благодаря общению с этими людьми. Перечислить всех их не представляется возможным, но я хочу выразить признательность Элизабет Бонд (Elizabeth Bond), Джону Полу Догину (John Paul Doguin), Марку Эшеру (Marc Esher), Дженн Ласситер (Jenn Lassiter) и Джессике Шафер (Jessica Schafer).

Весь процесс написания сопровождался музыкой, и эта книга могла не увидеть свет без прекрасных композиций, созданных Чаком Джонсоном (Chuck Johnson), Мэри Латтимор (Mary Lattimore), Макайей Маккрэвен (Makaya McCraven), Г. С. Шрейем (G. S. Schray), Сэмом Уилкесом (Sam Wilkes), Хироши Йошимурой (Hiroshi Yoshimura) и многими другими.

В заключение я хочу поблагодарить свою жену Эбби (Abbey) и наших детей Райли (Riley), Харрисона (Harrison) и Харлоу (Harlow), которым я зачастую не мог уделить достаточно внимания, так как был занят написанием книги. Спасибо, что терпели мои постоянные задержки в офисе и то, что даже будучи с вами, я не мог перестать думать о работе. Вы четверо являетесь мощнейшей мотивацией для всего, что я делаю.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Среда разработки

Джон Вуден (John Wooden), покойный тренер баскетбольной команды UCLA, считается одним из наиболее успешных тренеров всех времен. Его команда в течение двенадцати лет выиграла 10 национальных чемпионатов. В ее состав входили лучшие игроки, включая таких членов зала славы, как Лью Алсиндор (Lew Alcindor) (настоящее имя — Карим Абдул-Джаббар [Kareem Abdul-Jabbar]) и Билл Волтон (Bill Walton). Вуден имел обыкновение в первый день тренировки усаживать всех только что нанятых игроков — лучших в США среди всех старших школ — и учить их правильно надевать носки. В ответ на вопросы по этому поводу Вуден заявлял (https://oreil.ly/lnZkf), что «именно из мелочей складываются большие дела».

Шеф-повара описывают подготовку необходимых для приготовления блюд инструментов и ингредиентов выражением *mise en place*, что означает «все на своих местах». Благодаря учету всех мелких деталей такая подготовка позволяет им успешно справляться с готовкой во время повышенной загруженности. По аналогии с баскетбольным тренером, подготавливающим своих игроков, и поварами, готовящими блюда к обеденному наплыву клиентов, нам тоже стоит уделить время настройке среды разработки.

Для создания ее эффективного варианта вовсе не обязательно использовать дорогое ПО или самое производительное оборудование. Вообще я призываю вас начать с простого — задействовать открытое программное обеспечение (OSS) и постепенно добавлять к нему нужные инструменты. Хотя бегун и предпочитает определенный бренд кроссовок, а плотник может выбирать особенный молоток, для выработки таких предпочтений требуется время и опыт. Экспериментируйте с инструментами, наблюдайте за появлением новых и со временем вы создадите максимально удобную для вас среду разработки.

В этой главе мы сосредоточимся на установке текстового редактора, Node.js, Git, MongoDB и нескольких полезных JS-пакетов, а также определим место для терминального приложения. Возможно, у вас уже есть удобно настроенная среда разработки. Но мы тем не менее также установим несколько необходимых инструментов, которые будем использовать на протяжении всей книги. Даже

если вы, как и я, предпочитаете пропускать инструкции, то в данном случае я все же советую с ними ознакомиться.

Если в какой-то момент у вас возникнут сложности с пониманием материала книги, смело обращайтесь к ее сообществу через канал Spectrum по адресу spectrum.chat/jseverywhere.

Текстовый редактор

Текстовые редакторы во многом похожи на одежду: они нам необходимы, но личные предпочтения при этом могут сильно различаться. Некоторым нравятся простые и грамотно организованные капсулы, другие предпочитают яркий принт пейсли. Неверных решений тут не бывает, и вам следует использовать наиболее удобный для вас вариант.

Если у вас все еще нет фаворита, я очень рекомендую Visual Studio Code (https://code.visualstudio.com). Этот редактор с открытым исходным кодом доступен для платформ Mac, Windows и Linux. Он предлагает встроенные функции для упрощения разработки и легко изменяется при помощи расширений, предлагаемых сообществом. Кстати говоря, разработан он также на JavaScript.

Терминал

Если вы используете VSCode, то у него есть встроенный терминал. Для большинства задач по разработке этого вполне достаточно. Лично я предпочитаю использовать отдельный терминальный клиент, поскольку так мне проще управлять несколькими вкладками и использовать больше выделенного оконного пространства на компьютере. Предлагаю вам попробовать оба варианта и выбрать наиболее оптимальный.

Использование отдельного приложения терминала

Все операционные системы снабжены встроенным приложением терминала, который послужит отличной отправной точкой. В macOS он называется вполне себе очевидно: Terminal. В ОС Windows, начиная с Windows 7, аналогичная программа называется PowerShell. В дистрибутивах Linux название терминала может быть разным, но зачастую содержит слово Terminal.

Использование VSCode

Для получения доступа к терминалу в VSCode кликните Terminal → New Terminal, и перед вами откроется его окно. Командная строка будет располагаться в каталоге текущего проекта.

Навигация по файловой системе

После обнаружения терминала вам пригодится умение ориентироваться в файловой системе. Это можно делать при помощи команды `cd`, которая отвечает за смену каталога.

ИНСТРУКЦИИ КОМАНДНОЙ СТРОКИ

Инструкции терминала часто включают символы `$` или `>` в начале строки. Они используются только для указания на инструкцию, и копировать их не нужно. В данной книге я буду обозначать инструкции терминала знаком доллара (`$`). При их вводе в собственном терминале этот знак включать не надо.

Открыв приложение терминала, вы увидите командную строку с курсором, в которой можно вводить команды. По умолчанию вы находитесь в домашнем каталоге компьютера. Если вы еще этого не делали, то я рекомендую создать каталог Projects, который будет подкаталогом домашней директории. Этот каталог будет содержать все ваши проекты. Создать и перейти в него вы можете следующим образом:

```
# сначала вводите команду cd, которая обеспечит переход в корневую директорию:
$ cd
# затем, если вы еще не создавали каталог Projects, можете это сделать;
# для этого используйте следующую команду – она создаст Projects в виде
# подкаталога в корневой директории системы:
$ mkdir Projects
# наконец, вы можете войти в этот каталог:
$ cd Projects
```

В дальнейшем переходить в директорию Projects можно так:

```
$ cd # выполняет переход в корневую директорию
$ cd Projects
```

Теперь давайте предположим, что в директории Projects у вас есть каталог jseverywhere. Для перехода в него из Projects нужно ввести `cd jseverywhere`. Для возвращения на один каталог выше (в данном случае в Projects) нужно набрать `cd ..` (команда `cd`, за которой следуют две точки).

Все вместе это будет выглядеть так:

```
> $ cd # убедитесь, что вы в корневом каталоге
> $ cd Projects # для перехода из корневой директории в директорию Projects
/Projects > $ cd jseverywhere # для перехода из Projects в директорию
 jsevewehre
/Projects/jseverwhere > $ cd .. # для возвращения из jseverwhere в Projects
/Projects > $ # Командная строка находится в директории Projects
```

Если вы с такими вещами раньше не сталкивались, то потренируйтесь немного в перемещении между файлами, чтобы чувствовать себя более уверенно. Я выяс-

нил, что начинающие разработчики часто спотыкаются о сложности с файловой системой. Освоив эти действия, вы сможете гораздо увереннее организовывать собственные рабочие процессы.

Инструменты командной строки и Homebrew (только для Mac)

Некоторые утилиты командной строки доступны только пользователям macOS при установке Xcode. Вы же можете обойтись без самого Xcode, просто установив `xcode-select` через терминал. Для этого выполните следующую команду и следуйте подсказкам установщика:

```
$ xcode-select --install
```

Homebrew — это пакетный менеджер для macOS. Он позволяет устанавливать зависимости разработки вроде языков и баз данных так же просто, как и выполнять инструкции в командной строке. Если вы используете Mac, то он существенно упростит настройку среды разработки. Для установки Homebrew либо скопируйте и вставьте соответствующую команду с сайта brew.sh, либо наберите в одну строку следующее:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Node.js и NPM

Node.js (https://nodejs.org) — это «среда выполнения JavaScript, построенная на JS-движке V8, разработанном Google». Фактически же это означает, что Node является платформой, позволяющей разработчикам писать JS-код вне среды браузера. Node.js поставляется с NPM, пакетным менеджером, по умолчанию. NPM позволяет устанавливать в рамках вашего проекта тысячи библиотек и инструментов JavaScript.

УПРАВЛЕНИЕ ВЕРСИЯМИ NODE.JS

Если вы планируете управлять большим числом проектов Node, то можете столкнуться с необходимостью управления множеством версий этой среды на вашем компьютере. В таком случае я рекомендую для установки Node использовать его менеджер версий (NVM) (https://oreil.ly/fzBpO). NVM — это скрипт, который позволяет управлять несколькими активными версиями Node. Для пользователей Windows я рекомендую nvm-windows (https://oreil.ly/qJeej). Я не буду рассматривать управление версиями Node, но имейте в виду, что это очень полезный инструмент. Если для вас это первое знакомство с Node, то я советую продолжить следовать дальнейшим инструкциями для вашей системы.

Установка Node.js и NPM для macOS

Пользователи macOS могут установить Node.js и NPM, используя Homebrew. Для установки Node.js наберите в терминале следующую команду:

```
$ brew update
$ brew install node
```

Установив Node, откройте приложение терминала, чтобы убедиться в его работоспособности.

```
$ node --version
## Ожидаемый вывод v12.14.1, номер вашей версии может отличаться
$ npm --version
## Ожидаемый вывод 6.13.7, номер вашей версии может отличаться
```

Если после ввода указанных команд вы увидите номер версии, то я вас поздравляю — вы успешно установили Node и NPM для macOS!

Установка Node.js и NPM для Windows

Для Windows проще всего установить Node.js, посетив сайт nodejs.org и скачав установщик для вашей ОС.

Сначала перейдите на nodejs.org и установите LTS-версию (на момент написания книги 12.14.1), следуя шагам установки для вашей операционной системы. Установив Node, откройте терминал для проверки его работоспособности.

```
$ node --version
## Ожидаемый вывод v12.14.1, номер вашей версии может отличаться
$ npm --version
## Ожидаемый вывод 6.13.7, номер вашей версии может отличаться
```

ЧТО ТАКОЕ LTS?

LTS — это аббревиатура от long-term support («долгосрочная поддержка»), то есть Node.js Foundation обязуется предоставлять поддержку и обновления безопасности для этого основного номера версии (в нашем случае 12.x). Стандартный период поддержки длится три года после изначального релиза версии. В Node.js релизы с четными номерами являются LTS-версиями. Я рекомендую для разработки приложений использовать релиз именно с четным номером.

Если после ввода этих команд вы увидите номер версии, то я вас поздравляю — вы успешно установили Node и NPM для Windows!

MongoDB

MongoDB — это база данных, которую мы будем использовать в процессе разработки нашего API. Mongo весьма популярна при работе с Node.js, поскольку

воспринимает данные в виде документов JSON (JavaScript Object Notation), что существенно облегчает JS-разработчикам их использование.

ОФИЦИАЛЬНАЯ ДОКУМЕНТАЦИЯ ПО УСТАНОВКЕ MONGODB

Документация MongoDB предлагает регулярно обновляемое руководство по установке MongoDB Community Edition в различных операционных системах. Если у вас в процессе установки возникнут сложности, я рекомендую обратиться к ней по ссылке docs.mongodb.com/manual/administration/install-community.

Установка и запуск MongoDB для macOS

Сначала установите ее с помощью Homebrew:

```
$ brew update
$ brew tap mongodb/brew
$ brew install mongodb-community@4.2
```

Запустить же MongoDB можно в качестве службы macOS:

```
$ brew services start mongodb-community
```

Эта команда запустит службу MongoDB, которая продолжит работу в фоновом режиме. Обратите внимание, что каждый раз, когда вы перезапускаете компьютер и планируете заниматься разработкой с применением Mongo, вам может потребоваться повторно перезапускать службу MongoDB. Чтобы убедиться в правильности ее установки и работоспособности, наберите `ps -ef | grep mongod` в терминале. Эта команда выведет список всех текущих процессов Mongo.

Установка и запуск MongoDB для Windows

Сначала загрузите установщик из центра загрузки MongoDB (https://oreil.ly/XNQj6). После этого запустите его, следуя подсказкам мастера установки. Я советую выбрать вариант Complete с конфигурацией MongoDB как службы. Все остальные значения можно оставить по умолчанию.

После завершения установки может потребоваться создать директорию, в которую Mongo будет записывать данные. Выполните через терминал следующие команды:

```
$ cd C:\
$ md "\data\db"
```

Для проверки успешности установки и запуска службы Mongo:

1. Откройте консоль служб Windows.
2. Найдите службу MongoDB.

3. Кликните по ней правой кнопкой мыши.
4. Выберите Запуск.

Помните, что каждый раз, когда вы перезапускаете компьютер и планируете вести разработку, используя Mongo, вам может потребоваться повторно запустить службу MongoDB.

Git

Git — это самое популярное ПО для контроля версий, которое позволяет вам выполнять такие действия, как копирование репозиториев кода, слияние одного кода с другим и создание независимых веток собственного кода. Git также поможет с клонированием репозиториев с примерами кода из этой книги, предоставив возможность просто копировать каталоги со всем содержимым.

В некоторых ОС Git может быть установлен по умолчанию. Для проверки его наличия введите в окно терминала следующее:

```
$ git --version
```

Если в ответ вы получите номер, опять же поздравляю — Git у вас установлен. Если нет, установите его с сайта git-scm.com либо используйте Homebrew для macOS. Закончив установку, еще раз наберите `git --version` в терминале, чтобы проверить его работоспособность.

Expo

Expo — это пакет инструментов, упрощающий начальную загрузку и разработку проектов под iOS/Android с помощью React Native. Нам понадобится установить инструмент командной строки Expo и приложение Expo (не обязательно, но рекомендуется) для iOS или Android. Это мы подробнее рассмотрим в разделе, посвященном мобильным приложениям, но если вы хотите забежать вперед, то можете ознакомиться с деталями на сайте expo.io. Для установки инструментов командной строки введите в терминале следующую команду:

```
npm install -g expo-cli
```

Использование флага `-g` сделает инструмент `expo-cli` доступным глобально для Node.js, установленной на вашей машине.

Чтобы установить мобильные приложения Expo, посетите App Store или Google Play.

Prettier

Prettier — это инструмент для форматирования кода, поддерживающий ряд языков, включая JavaScript, HTML, CSS, GraphQL и Markdown. Он упрощает следование основным правилам форматирования. Это означает, что при выполнении его команды код автоматически форматируется в соответствии со стандартным набором наилучших правил. Более того, вы можете настроить редактор на выполнение этого действия в автоматическом режиме при сохранении файла. Благодаря этому вы забудете про несогласованные пробелы и разномастные кавычки.

Я рекомендую установить Prettier глобально и настролить плагин для редактора. Для установки перейдите в командную строку и наберите:

```
npm install -g prettier
```

После установки Prettier посетите Prettier.io, где вы сможете найти плагин для своего текстового редактора. После его установки я рекомендую добавить в его файл конфигурации следующие настройки:

```
"editor.formatOnSave": true,
"prettier.requireConfig": true
```

Благодаря этим настройкам файлы будут автоматически форматироваться при сохранении, если файл конфигурации .prettierrc будет присутствовать внутри проекта. Теперь при наличии этого файла ваш редактор всегда будет переформатировать код в соответствии с условиями проекта. Все проекты, описанные в рамках данной книги, будут содержать файл .prettierrc.

ESLint

ESLint — это линтер кода для JavaScript. Линтер отличается от средства форматирования вроде Prettier тем, что дополнительно проверяет код на такие отклонения в качестве, как неиспользуемые переменные, бесконечные циклы и недостижимый код, падающий после возврата. Как и в случае с Prettier, я рекомендую установить для вашего любимого редактора плагин ESLint. Это позволит получать предупреждение об ошибках прямо в процессе написания кода. Список плагинов для редактора вы можете найти на сайте ESLint (https://oreil.ly/H3Zao).

Также аналогично Prettier проекты могут определять желаемые правила ESLint внутри файла с расширением .eslintrc. Это даст обслуживающим проект разработчикам возможность тщательно контролировать предпочтения в коде и автоматизировать внедрение стандартов кодирования. Каждый проект, описанный в данной книге, будет включать хоть и полезный, но весьма вольный набор

правил ESLint, цель которого — помочь вам избежать самых распространенных ошибок.

Наводим красоту

Это не обязательно, но мне приятнее заниматься программированием, когда все выглядит эстетически. Ничего не могу с этим поделать. У меня есть диплом в области искусств, и вырос я с любовью к теме Dracula (https://draculatheme.com), которая представляет собой цветовую схему, доступную почти для каждого текстового редактора и терминала наряду с гарнитурой Source Code Pro от Adobe (https://oreil.ly/PktVn).

Итоги

В этой главе мы настроили на компьютере рабочую и гибкую среду разработки JavaScript. Одно из величайших удовольствий в программировании — персонализация собственной среды. Я призываю вас экспериментировать с темами, цветами и инструментами, чтобы сделать ее индивидуальной. В следующем разделе книги мы уже применим эту среду на деле, разработав приложение API.

ГЛАВА 2

Введение в API

Вообразите себя сидящим в кабинке небольшого местного ресторана, где вы решили заказать сэндвич. Официант записывает ваш заказ на листке бумаги и передает его повару. Повар читает заказ, берет конкретные ингредиенты для приготовления и передает готовое блюдо официанту. Затем официант приносит его вам. Если после этого вы захотите десерт, процесс повторится.

Интерфейс программирования приложения (application programming interface, API) представляет собой набор спецификаций, позволяющий одной компьютерной программе взаимодействовать с другой. Веб-API во многом работает по похожей на заказ сэндвича схеме. Клиент запрашивает данные, которые отправляются приложению веб-сервера через протокол передачи гипертекста (HyperText Transfer Protocol, HTTP); приложение получает эти запросы и обрабатывает данные, которые затем отправляются обратно клиенту также через HTTP.

В этой главе мы изучим обширную тему веб-API и начнем разработку с клонирования стартового проекта API на нашу локальную машину. Но для начала я предлагаю рассмотреть требования приложения, которое будем создавать.

Что мы создаем

На протяжении книги мы будем создавать приложение для социальных заметок под названием Notedly. Его пользователи будут иметь возможность создать аккаунт, писать заметки в виде простого текста или Markdown, редактировать их, просматривать ленту и «избранные» заметки других пользователей. В текущем разделе книги мы будем разрабатывать API для поддержки этого приложения.

В нашем API пользователи смогут:

- создавать заметки, а также читать, обновлять и удалять их;
- просматривать ленту заметок, созданных другими пользователями, и читать их отдельные заметки без возможности обновления или удаления;

- создавать аккаунт, авторизовываться и выходить из системы;
- просматривать информацию своего профиля, а также публичную информацию профилей других пользователей;
- добавлять в «избранное» заметки других пользователей и составлять из них список.

MARKDOWN

Markdown — это популярный язык разметки, распространенный в сообществе программистов наряду с такими текстовыми приложениями, как iA Writer, Ulysses, Byword и др. Подробнее узнать о Markdown вы можете на сайте его руководства (https://www.markdownguide.org).

Звучит внушительно, но я буду разбивать реализацию этой функциональности на небольшие части. Научившись организовывать такие типы взаимодействия, вы сможете применять их для создания всех видов API.

Как мы будем это создавать

Для сборки API мы будем использовать язык запросов GraphQL API (https://graphql.org). GraphQL — это спецификация с открытым исходным кодом, изначально разработанная в Facebook в 2012 году. Ее плюс в том, что она позволяет клиенту запрашивать именно те данные, которые ему нужны, что существенно упрощает и снижает число запросов. Это также дает явное преимущество в производительности при отправке ответов на мобильные клиенты, ведь в этом случае тоже отправляются только необходимые данные. На протяжении книги мы много времени будем уделять изучению написания, разработки и использования GraphQL.

А КАК ЖЕ REST?

Если вы знакомы с терминологией веб-API, то наверняка слышали о REST (Representational State Transfer — «передача состояния представления») API. Архитектура REST до сих пор остается ведущим форматом для API. Эти API отличаются от GraphQL тем, что при обращении к серверу опираются на структуру URL и параметры запросов. Несмотря на то что REST вполне подходит для наших задач, простота GraphQL, надежность его инструментария и потенциальный прирост производительности при отправке ограниченного количества данных по Сети, на мой взгляд, делает этот язык запросов предпочтительным средством для современных платформ.

Начало

Прежде чем начать разработку, нам нужно сделать копию стартовых файлов проекта на компьютер. Исходный код проекта (https://oreil.ly/mYKmE) содержит

все сценарии и ссылки на сторонние библиотеки, которые понадобятся нам для разработки приложения. Для клонирования кода на локальный компьютер откройте терминал, перейдите в директорию, где хранятся проекты, выполните команду `git clone` для репозитория и установите зависимости командой `npm install`. Для организации всего кода этой книги будет также полезным создать каталог notedly:

```
$ cd Projects
$ mkdir notedly && cd notedly
$ git clone git@github.com:javascripteverywhere/api.git
$ cd api
$ npm install
```

УСТАНОВКА СТОРОННИХ ЗАВИСИМОСТЕЙ

Сделав копию стартового кода книги и выполнив `npm install` в директории, вы избегаете повторного выполнения этой команды для каждой отдельной сторонней зависимости.

Код структурирован следующим образом:

/src

Это директория, в которой вы будете вести разработку по ходу изучения книги.

/solutions

В этой директории содержатся решения для каждой главы. К ней вы можете обратиться в случае возникновения сложностей.

/final

В этой директории будет содержаться итоговый рабочий проект.

Теперь, когда у вас есть код на собственном компьютере, нужно сделать копию .env файла проекта. Этот файл служит для хранения относящейся к среде информации или таких секретов проектов, как URL базы данных, ID клиентов и пароли. Поэтому его никогда не следует проверять в системе контроля версий, так что вам понадобится собственная копия этого файла. Для ее получения, находясь в директории api, наберите в терминале следующее:

```
cp .env.example .env
```

Теперь вы должны увидеть файл .env в этой директории. Вам пока что не нужно ничего с ним делать, но позже по мере разработки бэкенда нашего API мы будем добавлять в этот файл информацию. Файл .gitignore, включенный в проект, гарантирует, что вы не сделаете коммит файла .env по неосторожности.

Я НЕ ВИЖУ ФАЙЛ .ENV!

По умолчанию операционные системы скрывают файлы, имена которых начинаются с точки, поскольку обычно такие файлы используются только системой, а не самими пользователями. Если вы не видите файл `.env`, попробуйте открыть директорию в текстовом редакторе. Файл должен найтись либо там, либо через проводник. Еще один способ — набрать в окне терминала команду `ls -a`, которая выведет список всех файлов в текущей рабочей директории.

Итоги

API предоставляют интерфейс для передачи данных из БД к приложениям, являясь таким образом опорой последних. Используя GraphQL, мы можем быстро разрабатывать современные масштабируемые приложения на основе API. В следующей главе мы начнем создание API с построения веб-сервера при помощи Node.js и Express.

ГЛАВА 3

Создание веб-приложения с помощью Node и Express

Прежде чем переходить к реализации API, мы создадим простое серверное приложение, которое ляжет в основу его бэкенда. Для этого мы будем использовать фреймворк Express.js (https://expressjs.com) — так называемый минималистический веб-фреймворк для Node.js. Он небогат возможностями, но очень легко поддается настройке. Express.js мы будем использовать в качестве основы для нашего сервера API, но его также можно задействовать для построения полнофункциональных серверных веб-приложений.

Пользовательские интерфейсы вроде сайтов и мобильных приложений связываются с веб-серверами, когда им нужен доступ к данным. Эти данные могут быть чем угодно, начиная от HTML, необходимого для отображения страницы в браузере и заканчивая результатами поиска по запросу пользователя. Интерфейс клиента общается с сервером через HTTP. Запрос данных отправляется от клиента через этот протокол к выполняемому на сервере веб-приложению. Это приложение, в свою очередь, обрабатывает полученный запрос и возвращает данные клиенту снова через HTTP.

В этой главе мы создадим небольшое серверное веб-приложение, которое послужит основой для нашего API. Для этого мы будем использовать фреймворк Express.js и получим в итоге простое веб-приложение, отправляющее базовый запрос.

Hello World

Теперь, когда у вас появилось понимание основ серверных веб-приложений, давайте перейдем к делу. Создайте в каталоге src файл с именем index.js и добавьте в него следующее:

```
const express = require('express');
const app = express();
```

```
app.get('/', (req, res) => res.send('Hello World'));
app.listen(4000, () => console.log('Listening on port 4000!'));
```

В этом примере нам сначала нужно установить зависимость `express` и при помощи импортированного модуля Express.js создать объект `app`. Затем мы используем метод `get` объекта `app`, чтобы дать команду приложению отправлять ответ «Hello World» при обращении пользователя к корневому URL (`/`). В завершение мы указываем, что приложение будет выполняться на порте 4000. Это позволит нам просматривать приложение локально по URL http://localhost:4000.

Теперь для запуска приложения наберите `node src/index.js` в терминале. После этого вы должны увидеть в нем запись `Listening on port 4000!`. Если так и произошло, у вас должна появиться возможность открыть окно браузера по ссылке http://localhost:4000 и увидеть результат, как на рис. 3.1.

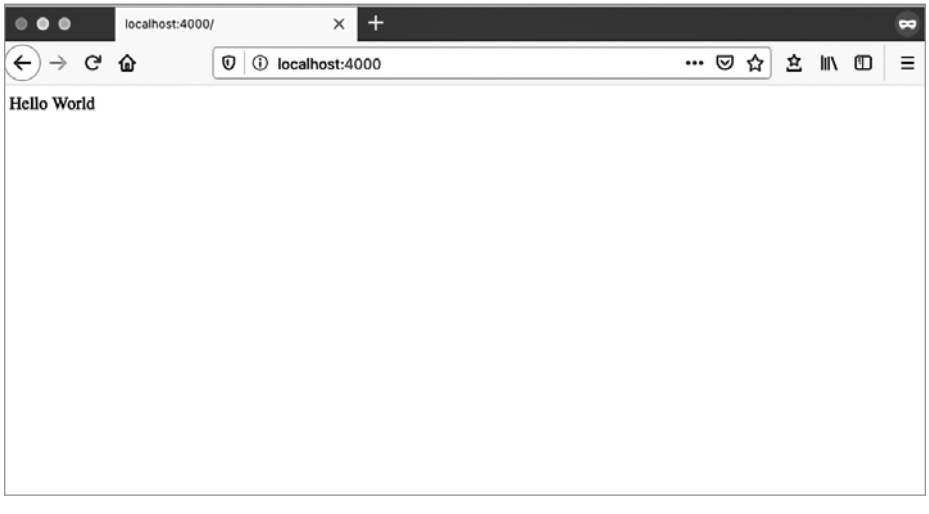

Рис. 3.1. Результат выполнения кода Hello World, отображенный в браузере

Nodemon

Теперь предположим, что результат этого примера не отражает наше настроение должным образом. Мы хотим изменить код так, чтобы он добавлял к ответу восклицательный знак. Сделайте это, изменив значение `res.send` на Hello World!!!. Вся строка должна теперь выглядеть так:

```
app.get('/', (req, res) => res.send('Hello World!!!'));
```

Если вы перейдете в браузер и обновите страницу, то заметите, что текст остался прежним. Так произошло, потому что для применения любых вносимых нами

в веб-сервер изменений требуется его перезапуск. Для этого перейдите обратно в терминал и остановите сервер нажатием Ctrl+C.

Для повторного запуска наберите `node index.js`. Вот теперь, вернувшись в браузер и обновив страницу, вы получите обновленный ответ.

Несложно представить, насколько быстро может надоесть остановка и перезапуск сервера для применения каждого изменения. К счастью, мы можем использовать пакет Node под названием `nodemon`, который будет делать это автоматически. Если вы взглянете на файл проекта package.json, то внутри объекта `scripts` увидите инструкцию `dev`, которая дает `nodemon` команду наблюдать за файлом index. js:

```
"scripts": {
  ...
  "dev": "nodemon src/index.js"
  ...
}
```

СЦЕНАРИИ PACKAGE.JSON

В объекте `scripts` присутствует несколько вспомогательных команд, с которыми мы постепенно познакомимся в последующих главах.

Теперь для запуска приложения из терминала наберите:

```
npm run dev
```

После перехода в браузер и обновления страницы вы увидите, что все осталось по-прежнему. Чтобы убедиться, что `nodemon` перезапускает сервер автоматически, давайте еще раз поменяем фразу в `res.send`:

```
res.send('Hello Web Server!!!')
```

Теперь вы сможете видеть внесенные изменения при обновлении страницы без ручной перезагрузки сервера.

Расширение опций порта

Пока что наше приложение работает на порте 4000. Такой вариант отлично подходит для локальной разработки, но при развертывании у нас должна быть возможность изменить номер порта. Давайте сейчас внесем это обновление и начнем с добавления переменной `Port`:

```
const port = process.env.PORT || 4000;
```

Это изменение позволит нам динамически устанавливать порт в среде Node и выполнять откат к порту 4000, если иной не будет установлен. Теперь давайте

настроим код `app.listen` так, чтобы он работал с этим обновлением, и используем литерал шаблона для регистрации правильного порта.

```
app.listen(port, () =>
  console.log(`Server running at http://localhost:${port}`)
);
```

Итоговый код теперь должен выглядеть так:

```
const express = require('express');

const app = express();
const port = process.env.PORT || 4000;

app.get('/', (req, res) => res.send('Hello World!!!'));

app.listen(port, () =>
  console.log(`Server running at http://localhost:${port}`)
);
```

Теперь у нас есть основа кода рабочего веб-сервера. Для проверки его работоспособности убедитесь, что в консоли отсутствуют ошибки, и перезагрузите окно браузера со ссылкой http://localhost:4000.

Итоги

Серверные веб-приложения являются основой разработки API. В этой главе мы создали простое веб-приложение при помощи фреймворка Express.js. При разработке приложений на основе Node вам доступен широкий арсенал фреймворков и инструментов. Express.js — отличный выбор из-за своей гибкости, поддержки сообщества и зрелости самого проекта. В следующей главе мы преобразуем наше веб-приложение в API.

ГЛАВА 4

Наш первый GraphQL API

Смею предположить, что вы человек, раз уж читаете эти строки. Будучи человеком, вы обладаете рядом интересов и пристрастий. У вас также есть члены семьи, друзья, знакомые, одноклассники и коллеги. У всех них, в свою очередь, тоже есть свои социальные связи, интересы и пристрастия. Некоторые из этих связей и интересов пересекаются, а некоторые — нет. В совокупности у каждого из нас есть связанный круг людей из нашей жизни.

GraphQL как раз и создавался для разрешения сложностей в таких запутанных типах взаимосвязей данных при разработке API. Написав GraphQL API, мы получаем возможность эффективно связывать данные, что снижает сложность и число запросов и в то же время позволяет нам передавать на клиент только нужные данные.

Не слишком ли это сложно для приложения, работающего с заметками? Может быть, так оно и звучит, но как вы увидите, инструменты и техники, предоставляемые экосистемой GraphQL JavaScript, не только делают возможными, но и упрощают любые виды разработки API.

В этой главе мы будем создавать GraphQL API, используя пакет `apollo-server-express`. Ради этого мы изучим фундаментальные темы, касающиеся GraphQL, напишем схему GraphQL, разработаем код для разрешения функций этой схемы и обратимся к нашему API через пользовательский интерфейс GraphQL Playground.

Превращение сервера в API (ну, вроде того)

Давайте начнем разработку API с преобразования нашего сервера Express в сервер GraphQL при помощи пакета `apollo-server-express`. Apollo Server (https://oreil.ly/1fNt3) — это открытая серверная библиотека GraphQL, работающая с большим числом серверных фреймворков Node.js, включая Express, Connect, Hapi и Koa. Он позволяет передавать данные из Node.js-приложения в виде GraphQL API и предоставляет полезные инструменты вроде GraphQL Playground — визуального помощника для взаимодействия с нашим API при разработке.

Чтобы написать API, мы изменим код веб-приложения из предыдущей главы. Давайте начнем с включения пакета `apollo-server-express`. Добавьте в начало файла src/index.js следующее:

```
const { ApolloServer, gql } = require('apollo-server-express');
```

Теперь, когда мы импортировали `apollo-server`, перейдем к настройке базового приложения GraphQL. Такие приложения состоят из двух основных компонентов: схемы определений типов и распознавателей, разрешающих запросы и мутации данных. Если вы ничего не поняли, это нормально. Мы реализуем ответ API «Hello World» и в процессе дальнейшей разработки будем подробнее изучать эти особенности GraphQL.

Для начала давайте построим базовую схему, которую будем хранить в переменной `typeDefs`. Эта схема будет описывать один `Query` («запрос») под названием `hello`, возвращающий строку:

```
// Построение схемы с использованием языка схем GraphQL
const typeDefs = gql`
  type Query {
    hello: String
  }
`;
```

Настроив схему, мы можем добавить распознаватель, который будет возвращать значение пользователю. Им будет простая функция, возвращающая строку «Hello World!»:

```
// Предоставляем функцию разрешения для полей схемы
const resolvers = {
  Query: {
    hello: () => 'Hello world!'
  }
};
```

Под конец мы интегрируем Apollo Server, который будет обслуживать наш GraphQL API. Для этого добавим некоторые специфичные для него настройки и промежуточное ПО, после чего обновим код `app.listen`:

```
// Настройка Apollo Server
const server = new ApolloServer({ typeDefs, resolvers });

// Применяем промежуточное ПО Apollo GraphQL и указываем путь к /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

После всего этого файл src/index.js должен выглядеть так:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

// Запускаем сервер на порте, указанном в файле .env, или на порте 4000
const port = process.env.PORT || 4000;

// Строим схему с помощью языка схем GraphQL
const typeDefs = gql`
  type Query {
    hello: String
  }
`;

// Предоставляем функции распознавания для полей схемы
const resolvers = {
  Query: {
    hello: () => 'Hello world!'
  }
};

const app = express();

// Настраиваем Apollo Server
const server = new ApolloServer({ typeDefs, resolvers });

// Применяем промежуточное ПО Apollo GraphQL и указываем путь к /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

Если вы оставили процесс nodemon запущенным, то можете переходить прямо в браузер. В противном случае вам потребуется запустить сервер, набрав в терминале npm run dev. Далее перейдите по ссылке http://localhost:4000/api: там будет GraphQL Playground (рис. 4.1). Это веб-приложение, идущее в комплекте с Apollo Server, представляет одно из важных преимуществ работы с GraphQL. В нем вы можете выполнять запросы и вносить изменения и при этом сразу видеть результаты. Кроме того, можно найти автоматически создаваемую документацию для API во вкладке Schema.

У синтаксиса GraphQL Playground по умолчанию темная тема. В книге же я буду использовать светлую из-за более высокой контратстности. Изменить это можно в настройках самой GraphQL Playground, доступ к которым можно получить, кликнув на иконке шестеренки.

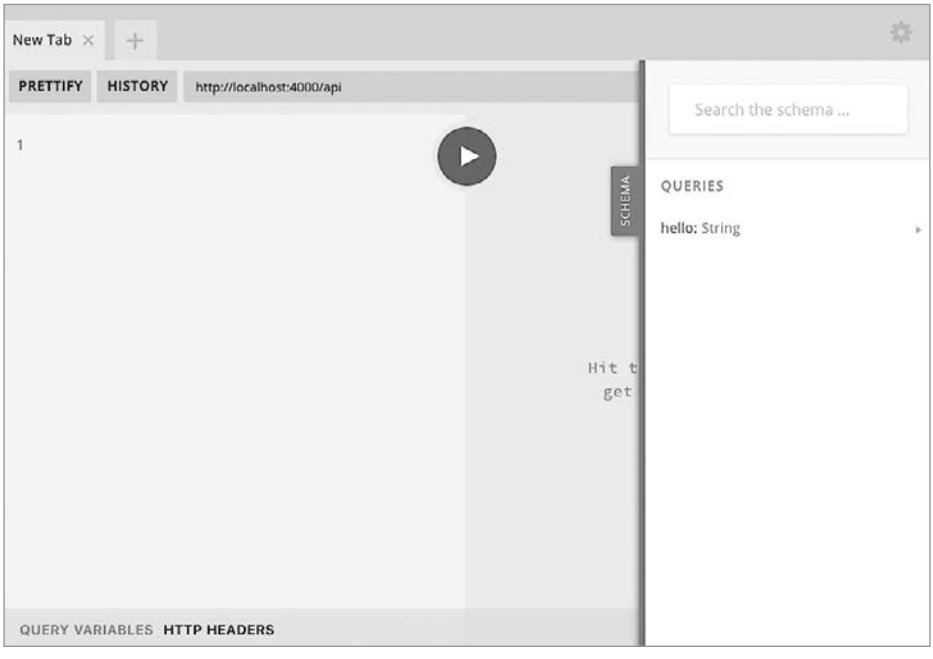

Рис. 4.1. GraphQL Playground

Рис. 4.2. Запрос hello

Теперь мы можем писать запрос к GraphQL API. Для этого наберите в GraphQL Playground следущее:

```
query {
  hello
}
```

Когда вы кликнете по кнопке Play, запрос должен вернуть следующее (рис. 4.2):

```
{
  "data": {
    "hello": "Hello world!"
  }
}
```

Вот и все! Теперь у нас есть рабочий GraphQL API, к которому мы обратились через GraphQL Playground. Наш API получает запрос `hello` и возвращает строку `Hello world!`. Но важнее то, что теперь у нас есть структура для построения полноценного API.

Основы GraphQL

В предыдущем разделе мы разработали наш первый API, но давайте ненадолго вернемся назад и взглянем на разные составляющие GraphQL API. Две из них — это схемы и распознаватели. Хорошенько разобравшись в этих компонентах, вы сможете более эффективно применять их в проектировании и разработке API.

Схемы

Схема — это письменное представление данных и взаимодействий. С ее помощью GraphQL обеспечивает соблюдение строгого плана нашего API, потому что API может возвращать данные и выполнять действия, которые определены в рамках этой схемы.

Основополагающим компонентом таких схем являются типы объектов. В предыдущем примере мы создали тип GraphQL-объекта `Query` с полем `hello`, который возвращал скалярный тип `String`. В GraphQL бывает пять встроенных скалярных типов:

`String`

Строка с кодировкой символов UTF-8.

`Boolean`

Значение true или false.

`Int`

32-битное целое число.

`Float`

Значение с плавающей точкой.

`ID`

Уникальный идентификатор.

С этими основными компонентами можно строить схемы для API. Начинать нужно с определения типа. Давайте представим, что создаем API для меню пиццерии. В этом случае мы можем определить тип GraphQL-схемы `Pizza` так:

```
type Pizza {
}
```

У каждой пиццы есть уникальный ID, размер (маленький, средний или большой), количество ломтиков и дополнительные начинки. В итоге схема `Pizza` может выглядеть так:

```
type Pizza {
  id: ID
  size: String
  slices: Int
  toppings: [String]
}
```

В этой схеме значения некоторых полей обязательны (`ID`, `size` и `slices`), другие же (в нашем случае `toppings`) — нет. Определить поле, которое требует ввод значения, мы можем с помощью восклицательного знака. Давайте перепишем схему, чтобы она отражала обязательные значения:

```
type Pizza {
  id: ID!
  size: String!
  slices: Int!
  toppings: [String]
}
```

В этой книге мы будем создавать простые схемы, которые позволят нам выполнять много операций, производимых в стандартном API. Если вас интересуют все опции схем GraphQL, советую обратиться к документации по этой теме (https://oreil.ly/DPT8C).

Распознаватели

Второй частью нашего GraphQL API являются распознаватели. Название говорит само за себя: они распознают данные, запрошенные пользователем API.

Мы напишем распознаватели, сначала определив их в схеме, а затем реализовав логику на JavaScript. Наш API будет содержать два типа распознавателей: запросы и мутации.

Запросы

С помощью них мы запрашиваем от API определенные данные в желаемом формате. В нашем гипотетическом API пиццерии мы можем написать один запрос, который будет возвращать полный список пицц в меню, и другой, возвращающий подробную информацию о конкретной пицце. Такой запрос будет возвращать объект с данными, запрошенными пользователем API. Запрос никогда не изменяет данные, а только обращается к ним.

Мутации

Мутацию используют, когда хотят изменить данные в API. В нашем примере с пиццей мы можем написать одну мутацию, изменяющую начинку для заданной пиццы, и другую, позволяющую нам скорректировать количество ломтиков. Как и запрос, мутация возвращает данные в форме объекта, представленного, как правило, конечным результатом выполненного действия.

Адаптация API

Теперь, когда вы хорошо понимаете компоненты GraphQL, пора заняться адаптацией нашего изначального кода API для приложения по работе с заметками. Начнем же мы этот процесс с написания кода для их чтения и создания.

Первое, что нам потребуется, — это немного данных, с которыми будет работать API. Давайте создадим массив объектов note, который будем использовать в качестве основных данных, предоставляемых нашим API. По мере развития проекта мы заменим это представление данных в памяти на БД. Сейчас же мы будем хранить эти данные в переменной `notes`. Каждая заметка в массиве будет представлена объектом с тремя свойствами: `id` (уникальный идентификатор), `content` (содержание) и `author` (автор):

```
let notes = [
  { id: '1', content: 'This is a note', author: 'Adam Scott' },
  { id: '2', content: 'This is another note', author: 'Harlow Everly' },
  { id: '3', content: 'Oh hey look, another note!', author: 'Riley Harrison' }
];
```

Далее, имея набор данных, мы адаптируем наш GraphQL API для работы с ними. Для начала давайте уделим внимание схеме. Она является GraphQL-представлением наших данных и способов взаимодействия с ними. Нам из-

вестно, что наши заметки будут запрашиваться и изменяться. Пока что они будут содержать поля ID, content и author. Давайте создадим соответствующий тип заметки внутри `typeDefs` схемы GraphQL, который будет представлять ее свойства внутри API:

```
type Note {
  id: ID!
  content: String!
  author: String!
}
```

А теперь давайте добавим запрос, который позволит нам извлекать список всех заметок. Для этого мы обновим тип Query, чтобы он включал запрос `notes`, который будет возвращать наш массив объектов:

```
type Query {
  hello: String!
  notes: [Note!]!
}
```

Далее мы можем обновить код распознавателя, чтобы он возвращал массив данных. Давайте изменим код `Query`, чтобы он включал следующий распознаватель `notes`, возвращающий объект с необработанными данными:

```
Query: {
    hello: () => 'Hello world!',
    notes: () => notes
  },
```

Если сейчас перейти на площадку GraphQL по ссылке http://localhost:4000/api, то можно протестировать запрос `notes`. Для этого наберите следующее:

```
query {
  notes {
    id
    content
    author
  }
}
```

После нажатия Play должен отобразиться объект `data`, содержащий массив данных (рис. 4.3).

Чтобы опробовать один из самых интересных аспектов GraphQL, мы можем удалить любое из запрашиваемых полей: например, `id` или `author`. Если это сделать, API вернет только конкретно запрошенные данные. Это позволяет клиенту, получающему эти данные, контролировать их количество при каждом запросе и ограничиваться только необходимыми (рис. 4.4).

Рис. 4.3. Запрос заметок

Рис. 4.4. Запрос заметок с данными только об их содержании

Теперь, когда мы можем запрашивать полный список заметок, давайте напишем код, который позволит нам запрашивать только одну из них. Можно представить, насколько это полезно с точки зрения пользовательского интерфейса. Для реализации нашего плана нужно будет запрашивать заметку с конкретным значением `id`. Для этого в GraphQL-схеме понадобится аргумент: он позволяет получателю API передавать в функцию распознавания конкретные значения, предоставляя ей необходимую информацию. Давайте добавим запрос `note`, который будет получать аргумент `id` с типом `ID`. Мы обновим наш объект `Query` в `typeDefs` так, чтобы он включал новый запрос `note`:

```
type Query {
  hello: String
  notes: [Note!]!
  note(id: ID!): Note!
}
```

После обновления схемы можно сделать так, чтобы распознаватель запросов возвращал запрашиваемую заметку. Для этого нам нужна возможность считывать значения аргументов пользователя API. К счастью, Apollo Server передает функциям распознавания следующие полезные параметры:

`parent`

Результат родительского запроса, который полезен при вложении запросов.

`args`

Аргументы, передаваемые пользователем в запросе.

`context`

Информация, которая передается от серверного приложения функциям распознавания и может включать, к примеру, данные о текущем пользователе или интересующее нас содержимое БД.

`info`

Информация о самом запросе.

Все это мы будем разбирать в рамках нашего кода по мере необходимости. При желании вы можете узнать подробнее об этих параметрах в документации Apollo Server (https://oreil.ly/l6mL4). А нам пока что нужна только информация, содержащаяся во втором параметре: `args`.

Запрос `note` будет получать в качестве аргумента `id` заметки и находить ее в нашем массиве объектов `notes`. Добавьте в код распознавателя запросов следующее:

```
note: (parent, args) => {
  return notes.find(note => note.id === args.id);
}
```

Теперь код распознавателя должен выглядеть так:

```
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: () => notes,
    note: (parent, args) => {
      return notes.find(note => note.id === args.id);
    }
  }
};
```

Для выполнения запроса давайте вернемся в браузер и перейдем на площадку GraphQL по ссылке http://localhost:4000/api. Теперь мы можем запросить заметку с конкретным `id` так:

```
query {
  note(id: "1") {
    id
    content
    author
  }
}
```

При выполнении этого запроса вы получите заметку с запрашиваемым значением `id`. Если вы попытаетесь запросить несуществующую заметку, то должны увидеть результат со значением `null`. Чтобы проверить это, попробуйте менять значение `id` для получения различных результатов.

Давайте завершим наш изначальный код API, добавив в него возможность пользователю создать новую заметку при помощи мутации GraphQL. Пока что мы жестко закодируем автора заметки. Начнем с добавления в нашу схему `typeDefs` типа `Mutation`, который будет вызывать `newNote`:

```
type Mutation {
  newNote(content: String!): Note!
}
```

Теперь напишем распознаватель мутации, который будет получать содержимое заметки в качестве аргумента, сохранять заметку в виде объекта и добавлять ее в наш массив `notes`. Для этого мы добавим в распознаватель объект `Mutation`. В сам же объект `Mutation` мы поместим функцию `newNote` с параметрами `parent` и `args`. В рамках функции мы возьмем аргумент `content` и создадим объект с ключами `id`, `content` и `author`. Как вы могли заметить, это соответствует текущей схеме заметки. Затем мы передадим этот объект в массив `notes` и вернем его. Возврат объекта позволяет мутации GraphQL получить ответ в нужном формате. Напишите следующий код:

```
Mutation: {
  newNote: (parent, args) => {
    let noteValue = {
```

```
      id: String(notes.length + 1),
      content: args.content,
      author: 'Adam Scott'
    };
    notes.push(noteValue);
    return noteValue;
  }
}
```

Теперь файл src/index.js будет выглядеть так:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

// Запускаем сервер на порте, указанном в файле .env, или на порте 4000
const port = process.env.PORT || 4000;

let notes = [
  { id: '1', content: 'This is a note', author: 'Adam Scott' },
  { id: '2', content: 'This is another note', author: 'Harlow Everly' },
  { id: '3', content: 'Oh hey look, another note!', author: 'Riley Harrison' }
];

// Строим схему, используя язык схем GraphQL
const typeDefs = gql`
  type Note {
    id: ID!
    content: String!
    author: String!
  }

  type Query {
    hello: String
    notes: [Note!]!
    note(id: ID!): Note!
  }

  type Mutation {
    newNote(content: String!): Note!
  }
`;

// Предоставляем функцию распознавания для полей схемы
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: () => notes,
    note: (parent, args) => {
      return notes.find(note => note.id === args.id);
    }
  },
  Mutation: {
    newNote: (parent, args) => {
      let noteValue = {
```

```
        id: String(notes.length + 1),
        content: args.content,
        author: 'Adam Scott'
      };
      notes.push(noteValue);
      return noteValue;
    }
  }
};

const app = express();

// Настраиваем Apollo Server
const server = new ApolloServer({ typeDefs, resolvers });

// Применяем промежуточное ПО Apollo GraphQL и указываем путь к /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

Обновив схему и распознаватель для принятия мутации, пора попробовать ее на GraphQL Playground по ссылке http://localhost:4000/api. Перейдя по ней, кликните по иконке + для создания новой вкладки и напишите мутацию:

```
mutation {
  newNote (content: "This is a mutant note!") {
   content
   id
   author
  }
}
```

После нажатия на Play вы должны получить ответ с content, ID и author новой заметки. Вы можете увидеть, что мутация сработала, повторно выполнив запрос `notes`. Для этого либо переключитесь обратно на вкладку GraphQL Playground, содержащую этот запрос, либо наберите следующее:

```
query {
  notes {
    content
    id
    author
  }
}
```

Теперь при выполнении этого запроса вы должны увидеть четыре заметки, включая недавно добавленную.

ХРАНЕНИЕ ДАННЫХ

Пока что мы храним данные в памяти. Это означает, что при каждом перезапуске сервера они будут стираться. Поэтому в следующей главе мы будем хранить данные уже в базе данных.

Мы успешно реализовали наши распознаватели запросов и мутаций, а также протестировали их в пользовательском интерфейсе GraphQL Playground.

Итоги

В этой главе мы собрали GraphQL API, используя модуль `apollo-server-express`. Теперь мы можем выполнять запросы и мутации в отношении объекта данных, находящегося в памяти. Эта настройка предоставляет нам прочный фундамент, на котором можно строить любой API. В следующей главе мы познакомимся с возможностью хранения данных, используя базу данных.

ГЛАВА 5

База данных

Еще мальчишкой я был одержим коллекционированием всякого рода спортивных карточек. Причем в этом занятии немало времени уделялось именно их сортировке. Я хранил звездных игроков в одной коробке, другая была отведена суперзвезде баскетбола Майклу Джордану, а остальные карточки я распределял по видам спорта и дополнительно классифицировал по командам. Такой метод организации позволил мне содержать все карточки в сохранности и с легкостью находить нужную. Тогда я и не подозревал, что такая система хранения является реальным эквивалентом базы данных. В своей сути БД позволяет нам сохранять и впоследствии извлекать нужную информацию.

Когда я впервые начал заниматься веб-разработкой, тема баз данных казалась мне пугающей. Я видел инструкции по запуску БД и вводу непонятных команд SQL, воспринимая все это как дополнительный уровень абстракции, который не умещался в моем сознании. К счастью, в итоге я во всем разобрался и больше не пугаюсь необходимости совмещения таблиц SQL. Поэтому если сейчас вы испытываете те же чувства, что и я в прошлом, то уверяю вас: сориентироваться в мире баз данных вполне возможно.

В этой книге в качестве БД мы будем использовать MongoDB (https://www.mongodb.com). Я выбрал ее из-за популярности в экосистеме Node.js, а еще потому, что она отлично подходит для новичков в этой области. Mongo хранит данные в «документах», которые работают аналогично JS-объектам. Это означает, что мы можем записывать и извлекать информацию в формате, знакомом любому JS-разработчику. Тем не менее, если вы предпочитаете другую базу данных вроде PostgreSQL, то рассматриваемые в книге темы без особых сложностей вполне можно перенести в систему любого типа.

Прежде чем мы сможем начать работу с Mongo, нужно убедиться, что MongoDB-сервер запущен локально. Это условие необходимо в процессе всей разработки. Для этого следуйте инструкциям для вашей системы из главы 1.

Начало работы с MongoDB

После запуска Mongo давайте разберемся, как взаимодействовать с ней напрямую из терминала, используя ее оболочку. Для начала откройте эту оболочку, набрав команду `mongo`:

```
$ mongo
```

После выполнения этой команды вы должны увидеть в терминале информацию об оболочке, подключении локального сервера и некоторые дополнительные сведения. Теперь мы можем взаимодействовать с MongoDB напрямую из терминала. Создавать, а также переключаться между БД можно при помощи команды `use`. Давайте создадим базу данных `learning`:

```
$ use learning
```

В моей коллекции, описанной в начале главы, я распределял карточки по разным коробкам. В MongoDB используется тот же принцип разделения, но уже по *коллекциям*. Коллекция — это то, как мы группируем схожие документы вместе. Например, приложение-блог может иметь отдельные коллекции для постов, пользователей и комментариев. Если бы мы сравнивали коллекцию с JS-объектом, то она была бы объектом верхнего уровня, а документы были бы отдельными объектами, расположенными внутри. Визуально это можно представить так:

```
collection: {
  document: {},
  document: {},
  document: {}.
  ...
}
```

На основе этой информации давайте создадим документ в нашей коллекции, расположенной в базе данных `learning`. Мы создадим коллекцию `pizza`, где будем хранить документы с видами пиццы. В оболочке MongoDB введите:

```
$ db.pizza.save({ type: "Cheese" })
```

Если все прошло успешно, то возвращаемый результат должен выглядеть так:

```
WriteResult({ "nInserted" : 1 })
```

Мы также можем сделать несколько записей в БД за один раз:

```
$ db.pizza.save([{type: "Veggie"}, {type: "Olive"}])
```

Теперь, когда мы записали ряд документов в базу данных, давайте их извлечем. Для этого мы будем использовать метод `find`. Чтобы увидеть все документы коллекции, выполните команду `find` с пустыми параметрами:

```
$ db.pizza.find()
```

Теперь мы должны увидеть в базе данных три записи. В дополнение к хранению данных MongoDB автоматически присваивает каждой записи уникальный ID. Результаты могут выглядеть, например, так:

```
{ "_id" : ObjectId("5c7528b223ab40938c7dc536"), "type" : "Cheese" }
{ "_id" : ObjectId("5c7529fa23ab40938c7dc53e"), "type" : "Veggie" }
{ "_id" : ObjectId("5c7529fa23ab40938c7dc53f"), "type" : "Olive" }
```

А еще можно находить отдельные документы как по значениям свойств, так и с помощью присвоенных базой данных ID:

```
$ db.pizza.find({ type: "Cheese" })
$ db.pizza.find({ _id: ObjectId("A DOCUMENT ID HERE") })
```

Но нам нужна не только возможность находить документы, но и способ их обновлять. Для этого мы можем использовать метод `update`, который принимает в качестве первого параметра документ для внесения изменения, а в качестве второго — само изменение. Давайте обновим нашу пиццу `Veggie`, сделав ее `Mushroom`:

```
$ db.pizza.update({ type: "Veggie" }, { type: "Mushroom" })
```

Теперь, если мы выполним `db.pizza.find()`, то увидим, что документ был обновлен:

```
{ "_id" : ObjectId("5c7528b223ab40938c7dc536"), "type" : "Cheese" }
{ "_id" : ObjectId("5c7529fa23ab40938c7dc53e"), "type" : "Mushroom" }
{ "_id" : ObjectId("5c7529fa23ab40938c7dc53f"), "type" : "Olive" }
```

Мы можем не только обновлять, но и удалять документ, используя для этого метод `remove`. Давайте удалим из базы данных грибную пиццу (mushroom):

```
$ db.pizza.remove({ type: "Mushroom" })
```

Теперь при запросе `db.pizza.find()` видно только две записи в коллекции. Если мы решим, что больше не хотим хранить данные в коллекции, то можем выполнить метод `remove` с пустым параметром объекта, что приведет к ее уничтожению:

```
$ db.pizza.remove({})
```

Мы успешно использовали оболочку MongoDB для создания базы данных, а также для добавления в коллекцию, обновления и удаления документов. Эти фундаментальные операции послужат прочным основанием в процессе интеграции базы данных в наш проект.

В разработке мы также можем обращаться к БД, используя оболочку MongoDB. Это может оказаться полезным для задач вроде отладки и ручного удаления/обновления записей.

Подключение MongoDB к приложению

Немного освоившись с использованием MongoDB из оболочки, давайте подключим ее к нашему приложению. Для этого мы будем использовать ODM Mongoose (https://mongoosejs.com). Mongoose — это библиотека, которая сокращает и оптимизирует рутинный код посредством моделирования на основе схем, из-за чего работать с MongoDB в Node.js-приложениях становится проще. Все верно — еще одна схема! Позже вы увидите, что как только мы определим схему базы данных, работа с MongoDB через Mongoose станет похожа на типы команд, которые мы писали в оболочке Mongo.

Сначала нам понадобится обновить файл .env, добавив URL локальной базы данных. Это позволит нам указывать URL БД в любой рабочей среде (например, в локальной разработке и в производстве). По умолчанию URL локального сервера MongoDB следующий: mongodb://localhost:27017. К нему мы и добавим имя нашей базы данных. Итак, в файле .env определим переменную DB_HOST с URL экземпляра БД Mongo таким образом:

```
DB_HOST=mongodb://localhost:27017/notedly
```

Следующим шагом будет подключение базы данных к приложению. Давайте напишем код, который будет выполнять это подключение при запуске приложения. Для этого сначала создадим в директории src файл db.js. В нем мы пропишем код для подключения БД, а также добавим функцию `close` для закрытия этого подключения, что пригодится при тестировании приложения.

В src/db.js введите:

```
// Затребуем библиотеку mongoose
const mongoose = require('mongoose');

module.exports = {
  connect: DB_HOST => {
    // Используем обновленный парсер строки URL драйвера Mongo
    mongoose.set('useNewUrlParser', true);
    // Поставим findOneAndUpdate () вместо findAndModify ()
    mongoose.set('useFindAndModify', false);
    // Поставим createIndex () вместо sureIndex ()
    mongoose.set('useCreateIndex', true);
    // Используем новый механизм обнаружения и мониторинга серверов
    mongoose.set('useUnifiedTopology', true);
    // Подключаемся к БД
    mongoose.connect(DB_HOST);
    // Выводим ошибку при неуспешном подключении
    mongoose.connection.on('error', err => {
      console.error(err);
      console.log(
        'MongoDB connection error. Please make sure MongoDB is running.'
      );
```

```
      process.exit();
    });
  },

  close: () => {
    mongoose.connection.close();
  }
};
```

Теперь обновим src/index.js для вызова этого подключения. Чтобы это сделать, сначала импортируем конфигурацию .env, а также файл db.js. В начале импортируемых файлов введите следующее:

```
require('dotenv').config();
const db = require('./db');
```

Мне нравится хранить значение DB_HOST, определенное в файле .env, в виде переменной. Добавьте эту переменную сразу под определением переменной port:

```
const DB_HOST = process.env.DB_HOST;
```

Затем мы можем вызвать подключение, добавив в файл src/index.js следующее:

```
db.connect(DB_HOST);
```

Теперь файл src/index.js будет выглядеть так:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');
require('dotenv').config();

const db = require('./db');

// Запускаем сервер на порте, указанном в файле .env, или на порте 4000
const port = process.env.PORT || 4000;
// Сохраняем значение DB_HOST в виде переменной
const DB_HOST = process.env.DB_HOST;

let notes = [
  {
    id: '1',
    content: 'This is a note',
    author: 'Adam Scott'
  },
  {
    id: '2',
    content: 'This is another note',
    author: 'Harlow Everly'
  },
  {
    id: '3',
    content: 'Oh hey look, another note!',
```

```
    author: 'Riley Harrison'
  }
];

// Строим схему, используя язык схем GraphQL
const typeDefs = gql`
  type Note {
    id: ID
    content: String
    author: String
  }

  type Query {
    hello: String
    notes: [Note]
    note(id: ID): Note
  }

  type Mutation {
    newNote(content: String!): Note
  }
`;

// Предоставляем функцию распознавания для полей схемы
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: () => notes,
    note: (parent, args) => {
      return notes.find(note => note.id === args.id);
    }
  },
  Mutation: {
    newNote: (parent, args) => {
      let noteValue = {
        id: notes.length + 1,
        content: args.content,
        author: 'Adam Scott'
      };
      notes.push(noteValue);
      return noteValue;
    }
  }
};

const app = express();

// Подключаем БД
db.connect(DB_HOST);

// Настраиваем Apollo Server
const server = new ApolloServer({ typeDefs, resolvers });
```

```
// Применяем промежуточное ПО Apollo GraphQL и указываем путь к /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

Несмотря на то что фактически функциональность не изменилась, при выполнении `npm run dev` приложение должно успешно подключиться к базе данных и запуститься без ошибок.

Чтение и запись данных

Теперь, когда мы можем подключаться к БД, давайте напишем код для чтения и записи данных в рамках приложения. Mongoose позволяет определить, как данные будут храниться в базе в виде JS-объекта, и мы сможем сохранять и работать с теми данными, которые подходят под эту структуру модели. Учитывая все это, давайте создадим объект, называемый Mongoose-схемой.

Сначала создайте в директории src каталог models, где будет храниться этот файл-схема. Далее в этом каталоге создайте файл note.js. Теперь мы перейдем к определению базовой настройки этого файла src/models/note.js:

```
// Запросим библиотеку mongoose
const mongoose = require('mongoose');

// Определяем схему БД заметки
const noteSchema = new mongoose.Schema();

// Определяем модель 'Note' со схемой
const Note = mongoose.model('Note', noteSchema);
// Экспортируем модуль
module.exports = Note;
```

Далее мы определим схему в переменной `noteSchema`. Аналогично примеру с хранением данных в памяти наша текущая схема пока что будет включать содержимое заметки наряду с жестко закодированной строкой, представляющей автора. Мы также включим опцию добавления временных меток, которые будут автоматически сохраняться при создании и изменении заметки. Добавлять функциональность в схему мы будем по мере продвижения.

Структура Mongoose-схемы будет следующей:

```
// Определяем схему БД заметки
const noteSchema = new mongoose.Schema(
  {
```

```
    content: {
     type: String,
      required: true
    },
    author: {
      type: String,
      required: true
    }
  },
  {
    // Присваиваем поля createdAt и updatedAt с типом данных
    timestamps: true
  }
);
```

ПОСТОЯНСТВО ДАННЫХ

В процессе разработки мы будем обновлять и изменять модель данных, иногда удаляя все данные из БД. Так что я бы не советовал использовать этот API для хранения важной информации вроде школьных записей, списка дней рождения друзей или адресов ваших любимых пиццерий.

В целом файл src/models/note.js теперь должен выглядеть так:

```
// Запрашиваем библиотеку mongoose
const mongoose = require('mongoose');

// Определяем схему БД заметки
const noteSchema = new mongoose.Schema(
  {
    content: {
      type: String,
      required: true
    },
    author: {
      type: String,
      required: true
    }
  },
  {
    // Присваиваем поля createdAt и updatedAt с типом данных
    timestamps: true
  }
);

// Определяем модель 'Note'со схемой
const Note = mongoose.model('Note', noteSchema);
// Экспортируем модуль
module.exports = Note;
```

Чтобы упростить импорт модели в наше приложение Apollo Server Express, мы добавим в директорию src/models файл index.js. Так мы объединим наши модели в один JS-модуль. Несмотря на то что это необязательно, я считаю, что следовать этому паттерну по мере роста приложений и моделей баз данных весьма полезно. В файл src/models/index.js мы импортируем модель заметки и добавим ее в объект `models` для экспорта:

```
const Note = require('./note');

const models = {
  Note
};

module.exports = models;
```

Теперь мы можем ввести модели БД в код приложения, импортировав их в файл src/index.js:

```
const models = require('./models');
```

Завершив импорт кода моделей, мы можем адаптировать распознаватели для сохранения и считывания из базы данных, а не из переменной в памяти. Для этого мы перепишем запрос `notes` на получение заметок из БД, используя уже знакомый нам MongoDB-метод `find`:

```
notes: async () => {
  return await models.Note.find();
},
```

При запущенном сервере мы можем перейти на GraphQL Playground в браузере и выполнить запрос `notes`:

```
query {
  notes {
    content
    id
    author
  }
}
```

Ожидаемым результатом будет пустой массив, поскольку нам еще нужно добавить в БД хоть какие-то данные (рис. 5.1):

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

Рис. 5.1. Запрос notes

Чтобы мутация `newNote` добавляла заметки в БД, мы добавляем в нее метод `create` нашей модели MongoDB, который будет принимать объект. Начнем же мы с того, что жестко закодируем имя автора.

```
newNote: async (parent, args) => {
  return await models.Note.create({
    content: args.content,
    author: 'Adam Scott'
  });
}
```

Теперь мы можем отправиться на площадку GraphQL и написать мутацию, которая будет добавлять заметку в нашу базу данных.

```
mutation {
  newNote (content: "This is a note in our database!") {
   content
   author
   id
  }
}
```

Эта мутация будет возвращать новую заметку с содержимым, помещенным нами в аргумент, именем автора, а также ID, сгенерированным MongoDB (рис. 5.2).

Если теперь повторить запрос `notes`, то мы увидим извлечение нашей заметки из БД (рис. 5.3).

Последним шагом будет переписывание запроса `notes` для получения из базы данных конкретной заметки по уникальному ID, присваиваемому MongoDB каждой записи. Для этого мы используем Mongoose-метод findbyId:

```
note: async (parent, args) => {
  return await models.Note.findById(args.id);
}
```

Рис. 5.2. Мутация создает новую заметку в БД

Рис. 5.3. Запрос notes возвращает данные из БД

Теперь для извлечения из БД конкретной заметки мы можем написать запрос, используя уникальный ID, который находится в запросе `notes` или мутации `newNote`. Для этого напишем запрос `note` с аргументом `id` (рис. 5.4):

```
query {
  note(id: "5c7bff794d66461e1e970ed3") {
    id
    content
    author
  }
}
```

ID ВАШЕЙ ЗАМЕТКИ

ID, использованный в предыдущем примере, уникален для моей локальной БД. Убедитесь, что копируете ID из собственного запроса или результата мутации.

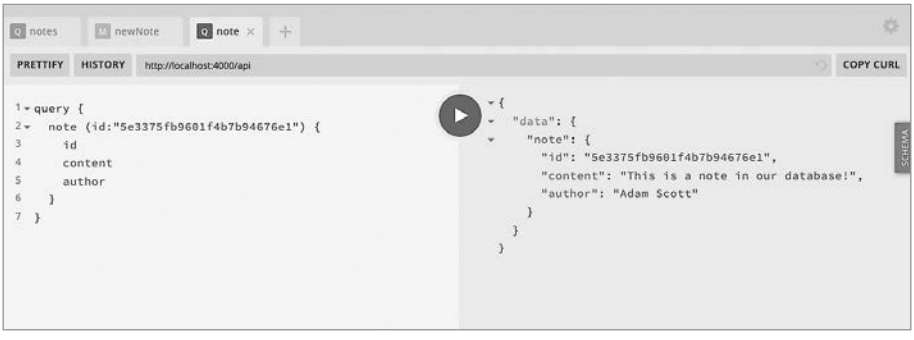

Рис. 5.4. Запрос конкретной заметки

В итоге файл src/index.js будет выглядеть так:

```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');
require('dotenv').config();

const db = require('./db');
const models = require('./models');
// Запускаем сервер на порте, указанном в файле .env, или на порте 4000
const port = process.env.PORT || 4000;
const DB_HOST = process.env.DB_HOST;

// Строим схему, используя язык схем GraphQL
const typeDefs = gql`
  type Note {
    id: ID
    content: String
    author: String
  }

  type Query {
    hello: String
    notes: [Note]
    note(id: ID): Note
  }

  type Mutation {
    newNote(content: String!): Note
  }
`;

// Предоставляем функцию распознавания для полей схемы
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
    notes: async () => {
```

```
      return await models.Note.find();
    },
    note: async (parent, args) => {
      return await models.Note.findById(args.id);
    }
  },
  Mutation: {
    newNote: async (parent, args) => {
      return await models.Note.create({
        content: args.content,
        author: 'Adam Scott'
      });
    }
  }
};

const app = express();

db.connect(DB_HOST);

// Настраиваем Apollo Server
const server = new ApolloServer({ typeDefs, resolvers });

// Применяем промежуточное ПО Apollo GraphQL и указываем путь к /api
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

Теперь мы можем считывать и записывать данные в БД с помощью нашего GraphQL API! Попробуйте добавить дополнительные заметки, получить их полный список с помощью запроса `notes` и просмотреть содержимое отдельных заметок через `note`.

Итоги

В этой главе вы научились пользоваться MongoDB и библиотеками Mongoose с нашим API. БД вроде MongoDB позволяют безопасно хранить и извлекать данные приложения. Библиотеки для моделирования объектов вроде Mongoose в свою очередь упрощают работу с БД, предоставляя инструменты для запросов и проверки данных. В следующей главе мы дополним API полноценной функциональностью CRUD для работы с содержимым нашей базы данных.

ГЛАВА 6

Операции CRUD

Когда я впервые услышал выражение «CRUD-приложение», то ошибочно предположил, что оно делает что-то нехорошее или не заслуживающее доверия. Акроним CRUD звучит так, как будто описывает нечто, соскребаемое с подошвы ботинка (в английском языке слово crud обозначает целый ряд не слишком приятных вещей; в данном случае автор намекает на «дерьмо». — *Примеч. ред.*). На самом же деле он впервые был популяризован в ранние 1980-е годы автором работ в области IT британцем Джеймсом Мартином (James Martin), в отношении приложений, которые создают (create), считывают (read), обновляют (update) и удаляют (delete) данные. Несмотря на то что выражение находится в обиходе уже более четверти века, оно по-прежнему применимо ко многим современным приложениям. Рассмотрим, к примеру, те, с которыми вы взаимодействуете ежедневно — списки дел, электронные таблицы, системы управления контентом, текстовые редакторы, соцсети и ряд других. Велика вероятность, что многие из них работают в формате CRUD. Пользователь создает, считывает данные, а также может обновлять или удалять их.

Наше приложение Notedly также будет придерживаться шаблона CRUD. Пользователи смогут создавать, читать, обновлять и удалять собственные заметки. В этой главе мы реализуем важнейшую функциональность CRUD нашего API, подключив распознаватели и базу данных.

Разделение GraphQL-схемы и распознавателей

На данный момент файл src/index.js содержит код сервера Express/Apollo, а также схему API и распознаватели. Легко представить, как по мере роста базы кода все это может привести к нагромождению. Не дожидаясь этого, проведем небольшой рефакторинг, который разделит код схемы, распознавателей и сервера.

Для начала давайте перенесем схему GraphQL в отдельный файл. Создадим в каталоге src файл и назовем его src/schema.js, а затем переместим в него со-

держимое схемы, расположенное сейчас в переменной `typeDefs`. Для этого нам также потребуется импортировать язык схем `qql`, поставляемый с пакетом apollo-server-express, и экспортировать схему в качестве модуля, используя Node-метод `module.exports`. Параллельно с этим мы можем сразу удалить запрос hello, который в финальной версии приложения нам не потребуется:

```
const { gql } = require('apollo-server-express');

module.exports = gql`
  type Note {
    id: ID!
    content: String!
    author: String!
  }

  type Query {
    notes: [Note!]!
    note(id: ID!): Note!
  }

  type Mutation {
    newNote(content: String!): Note!
  }
`;
```

Теперь мы можем обновить src/index.js, чтобы использовать внешний файл схемы, импортировав его и удалив импорт `gql` из `apollo-server-express` следующим образом:

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

const typeDefs = require('./schema');
```

Изолировав нашу GraphQL-схему в отдельный файл, сделаем то же самое с кодом распознавателя. Он будет включать большую часть логики API, поэтому для начала мы создадим каталог для его размещения и назовем его resolvers. В директории src/resolvers начнем с создания трех файлов: src/resolvers/index.js, src/resolvers/query.js и src/resolvers/mutation.js. Аналогично шаблону, которому мы следовали в моделях базы данных, файл src/resolvers/index.js будет использоваться для импорта кода распознавателей в один экспортируемый модуль. Настройте этот файл следующим образом:

```
const Query = require('./query');
const Mutation = require('./mutation');

module.exports = {
  Query,
  Mutation
};
```

Теперь вы можете настроить файл src/resolvers/query.js для кода запроса API:

```
module.exports = {
  notes: async () => {
    return await models.Note.find()
  },
  note: async (parent, args) => {
    return await models.Note.findById(args.id);
  }
}
```

Затем переместите код мутации в файл src/resolvers/mutation.js:

```
module.exports = {
  newNote: async (parent, args) => {
    return await models.Note.create({
      content: args.content,
      author: 'Adam Scott'
    });
  }
}
```

Далее для импорта сервером кода распознавателей добавьте следующую строку в файл src/index.js:

```
const resolvers = require('./resolvers');
```

Завершающим шагом рефакторинга распознавателей будет их подключение к нашим моделям БД. Как вы могли заметить, модули распознавания ссылаются на эти модели, но не имеют к ним доступа. Чтобы это исправить, используем принцип, который в Apollo Server называется *context*: он позволяет нам при каждом запросе передавать конкретную информацию из кода сервера в отдельный распознаватель. Сейчас это может показаться лишним, но позже понадобится для включения в наше приложение аутентификации пользователей. Ради этого мы добавим в код настройки Apollo Server, находящийся в src/index.js, функцию `context`, которая будет возвращать модели базы данных:

```
// Настройка Apollo Server
const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: () => {
    // Добавление моделей БД в context
    return { models };
  }
});
```

Теперь, чтобы каждый распознаватель использовал этот контекст, мы добавим в каждую функцию третий параметр, `{models}`:

В файле src/resolvers/query.js проделайте следующее:

```
module.exports = {
  notes: async (parent, args, { models }) => {
    return await models.Note.find()
  },
  note: async (parent, args, { models }) => {
    return await models.Note.findById(args.id);
  }
}
```

Переместите код мутации в файл src/resolvers/mutation.js:

```
module.exports = {
  newNote: async (parent, args, { models }) => {
    return await models.Note.create({
      content: args.content,
      author: 'Adam Scott'
    });
  }
}
```

Теперь наш файл src/index.js будет упрощен следующим образом:

```
const express = require('express');
const { ApolloServer } = require('apollo-server-express');
require('dotenv').config();

// Импортируем локальные модули
const db = require('./db');
const models = require('./models');
const typeDefs = require('./schema');
const resolvers = require('./resolvers');

// Запускаем сервер на порте, указанном в файле .env, или на порте 4000
const port = process.env.PORT || 4000;
const DB_HOST = process.env.DB_HOST;

const app = express();

db.connect(DB_HOST);

// Настраиваем Apollo Server
Const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: () => {
    // Добавляем модели БД в context
    return { models };
  }
});

// Применяем промежуточное ПО Apollo GraphQL и указываем путь к /api
```

```
server.applyMiddleware({ app, path: '/api' });

app.listen({ port }, () =>
  console.log(
    `GraphQL Server running at http://localhost:${port}${server.graphqlPath}`
  )
);
```

Написание CRUD-схемы

Закончив с рефакторингом кода и повысив тем самым его гибкость, мы займемся реализацией CRUD-операций. Сейчас мы уже можем создавать и читать заметки — остается только реализовать функциональность их обновления и удаления. Для начала нам нужно обновить схему.

Поскольку операции обновления и удаления будут вносить изменения в данные, они считаются мутациями. Для обновления заметки потребуется аргумент ID, по которому мы будем находить ее вместе с обновленным контентом. После этого запрос на обновление будет возвращать обновленную заметку. В случае удаления API будет возвращать логическое значение `true`, информируя нас об успехе операции.

Обновите схему Mutation в файле src/schema.js следующим образом:

```
type Mutation {
  newNote(content: String!): Note!
  updateNote(id: ID!, content: String!): Note!
  deleteNote(id: ID!): Boolean!
}
```

Теперь наша схема готова к выполнению CRUD-операций.

CRUD-распознаватели

Завершив подготовку схемы, мы можем заняться обновлением распознавателей для удаления или обновления заметок. Давайте начнем с мутации `deleteNote`. Чтобы удалить заметку, мы будем использовать Mongoose-метод `findOneAndRemove` и передавать в него `id` элемента, который нужно удалить. Если этот элемент будет найден и удален, то клиенту будет возвращено значение `true`, в случае неудачного удаления — `false`.

Добавьте в объект `module.exports` файла src/resolvers/mutation.js следующее:

```
deleteNote: async (parent, { id }, { models }) => {
  try {
    await models.Note.findOneAndRemove({ _id: id});
    return true;
```

```
    } catch (err) {
      return false;
    }
  },
```

Теперь можно выполнять мутацию на площадке GraphQL. В новой вкладке пропишите следующую мутацию, убедившись, что используете ID одной из заметок собственной базы данных:

```
mutation {
  deleteNote(id: "5c7d1aacd960e03928804308")
}
```

В случае успешного удаления заметки вы получите ответ со значением `true`:

```
{
  "data": {
    "deleteNote": true
  }
}
```

Если вы передадите несуществующий ID, то получите ответ вида `"deleteNote": false`.

Добавив функциональность удаления, давайте напишем мутацию `updateNote`. Для этого мы используем Mongoose-метод `findOneAndUpdate`. Этот метод будет получать начальный параметр запроса для нахождения нужной заметки в БД, сопровождаемый вторым параметром, в котором мы зададим `$set` новое содержимое заметки. В завершение мы передадим третий параметр `new: true`, согласно которому база данных будет возвращать обновленную заметку.

Добавьте в объект `module.exports` файла src/resolvers.mutation.js следующее:

```
updateNote: async (parent, { content, id }, { models }) => {
  return await models.Note.findOneAndUpdate(
    {
      _id: id,
    },
    {
      $set: {
        content
      }
    },
    {
      new: true
    }
  );
},
```

Теперь мы можем посетить площадку и опробовать мутацию `updateNote`. В новой вкладке пропишите эту мутацию с параметрами `id` и `content`:

```
mutation {
  updateNote(
    id: "5c7d1f0a31191c4413edba9d",
    content: "This is an updated note!"
  ){
    id
    content
  }
}
```

Если мутация сработает как ожидается, то ответ GraphQL будет следующим:

```
{
  "data": {
    "updateNote": {
      "id": "5c7d1f0a31191c4413edba9d",
      "content": "This is an updated note!"
    }
  }
}
```

Если мы передадим неверный ID, то результат будет неудачным и мы получим внутреннюю ошибку сервера с сообщением `Error updating note`.

Теперь мы можем создавать, читать, обновлять и удалять заметки, то есть имеем в нашем API полноценную CRUD-функциональность.

Время и дата

При создании схемы БД мы запросили, чтобы Mongoose автоматически добавлял временные метки, сохраняя время создания и изменения записей в базе данных. Эта информация пригодится нам в приложении, поскольку позволит в рамках UI (пользовательского интерфейса) показывать, когда заметка была создана или последний раз редактировалась. Давайте добавим в нашу схему поля `createdAt` и `UpdatedAt`, чтобы иметь возможность возвращать эти значения.

Помните, что GraphQL допускает такие типы по умолчанию, как `String`, `Boolean`, `Int`, `Float` и `ID`? К сожалению, GraphQL не имеет встроенного скалярного типа дат. Мы можем использовать тип `String`, но тогда не сможем воспользоваться преимуществом проверки типов, предлагаемым GraphQL, которое гарантировало бы нам, что перед нами действительно дата и время. Вместо этого мы создадим пользовательский скалярный тип: он позволяет определять новый тип и проверять его в отношении каждого запроса и мутации, запрашивающих данные этого типа.

Давайте обновим нашу GraphQL-схему в файле src/schema.js, добавив пользовательский скаляр в верхнюю часть строкового литерала GQL:

```
module.exports = gql`
  scalar DateTime
  ...
`;
```

Теперь добавьте в тип `Note` поля `createdAt` и `updatedAt`:

```
type Note {
  id: ID!
  content: String!
  author: String!
  createdAt: DateTime!
  updatedAt: DateTime!
}
```

Последним шагом будет проверка этого нового типа. Хотя мы и можем прописать собственную проверку, в данном случае мы будем использовать пакет `graphql-iso-date` (https://oreil.ly/CtmP6). Для этого мы добавим проверку во все функции распознавания, запрашивающие значения с типом `DateTime`.

Импортируйте этот пакет в файл src/resolvers/index.js и добавьте значение `DateTime` в экспортированные распознаватели:

```
const Query = require('./query');
const Mutation = require('./mutation');
const { GraphQLDateTime } = require('graphql-iso-date');

module.exports = {
  Query,
  Mutation,
  DateTime: GraphQLDateTime
};
```

Если теперь перейти на GraphQL Playground и обновить страницу, то можно проверить, работают ли наши пользовательские типы, как планировалось. Если обратиться к схеме, то можно увидеть, что поля `createdAt` и `updatedAt` имеют тип `DateTime`. Как показано на рис. 6.1, документация для этого типа сообщает, что он является «строкой даты и времени в формате UTC».

Чтобы все это проверить, давайте напишем на площадке GraphQL мутацию `newNote`, включающую поля даты:

```
mutation {
  newNote (content: "This is a note with a custom type!") {
    content
    author
    id
    createdAt
    updatedAt
  }
}
```

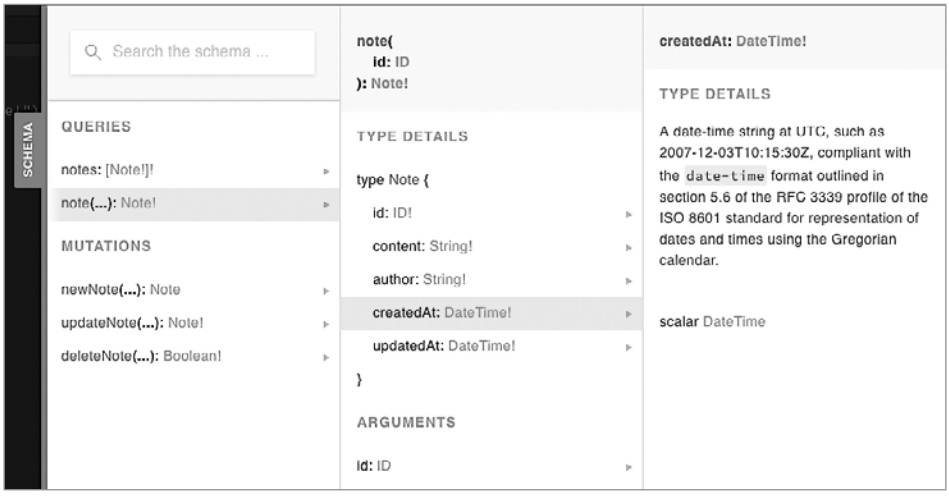

Рис. 6.1. Теперь в нашей схеме есть типы DateTime

Она вернет значения `createdAt` и `updatedAt` в виде даты в формате ISO. Если затем мы выполним мутацию `updateNote` в отношении той же заметки, то увидим значение `updatedAt`, отличающееся от даты `createdAt`.

За более подробной информацией касательно определения и проверки пользовательских скалярных типов я рекомендую обратиться к разделу *Custom scalars and enums* документации Apollo Server (https://oreil.ly/0rWAC).

Итоги

В этой главе мы добавили в наш API функциональность для создания, чтения, обновления и удаления данных (CRUD). CRUD-шаблон функциональности очень распространен среди многих приложений. Я призываю вас рассмотреть регулярно используемые вами приложения и проанализировать, как их данные могут вписываться в этот шаблон. В следующей главе мы добавим в наш API функциональность создания и аутентификации учетных записей пользователей.

ГЛАВА 7

Учетные записи пользователей и аутентификация

Представьте, что вы прогуливаетесь по темной аллее и направляетесь в Секретный клуб суперкрутых людей», чтобы стать его членом (если вы читаете эти строки, значит, точно этого заслуживаете). Возле потайной двери вас приветствует администратор и передает форму для заполнения: в ней вы должны указать свое имя и пароль, которые будут известны только вам и администратору.

Заполнив форму, вы передаете ее обратно администратору, и он отправляется в подсобную комнату клуба. Там он использует секретный ключ для шифрования вашего пароля и затем сохраняет его зашифрованную форму в закрытом хранилище файлов. Затем администратор штампует монету, на которой отпечатывается ваш уникальный членский ID. По возвращении в приемный кабинет администратор передает вам эту монету, которую вы прячете в карман. Теперь при каждом посещении для входа вам нужно только показать ее.

Описание этого взаимодействия может звучать в стиле низкобюджетного шпионского кино, зато очень напоминает процесс, который происходит при каждой регистрации пользователя в веб-приложении. В этой главе мы научимся строить GraphQL-мутации, которые дадут пользователю возможность создавать учетную запись и авторизовываться в нашем приложении. А еще мы узнаем, как шифровать пароль пользователя и возвращать токен, который затем можно будет использовать для верификации при взаимодействии с приложением.

Процесс аутентификации в приложении

Прежде чем начать, давайте схематично изобразим путь, по которому будут следовать пользователи при регистрации учетной записи или авторизации. Если вы до сих пор не поняли все описанные здесь принципы, то не волнуйтесь: разбирать их мы будем поэтапно. Для начала давайте рассмотрим процесс создания учетной записи:

1. Пользователь вводит свой имейл, имя пользователя и пароль в поля UI (например, на GraphQL Playground или в веб/мобильном приложении).
2. UI отправляет на наш сервер мутацию GraphQL с информацией о пользователе.
3. Сервер шифрует пароль и сохраняет эту информацию в базе данных.
4. Сервер возвращает в UI токен, содержащий ID пользователя.
5. UI хранит этот токен в течение определенного времени и отправляет его с каждым запросом к серверу, чтобы верифицировать пользователя.

Теперь рассмотрим процесс авторизации пользователя:

1. Пользователь вводит имейл или имя пользователя и пароль в поля UI.
2. UI отправляет GraphQL-мутацию с этой информацией на наш сервер.
3. Сервер расшифровывает пароль, хранящийся в БД, и сравнивает его с тем, который ввел пользователь.
4. Если пароли совпадают, сервер возвращает в UI токен, содержащий ID пользователя.
5. UI хранит этот токен в течение определенного времени и с каждым запросом отправляет его серверу.

Как вы видите, эти процессы очень похожи на историю с Секретным клубом. В этой главе мы сосредоточимся на реализации API-составляющих этих взаимодействий.

ПРОЦЕСС СБРОСА ПАРОЛЯ

Вы, наверное, заметили, что наше приложение не разрешает пользователям изменять их пароли. Мы могли бы добавить такую возможность с помощью всего одного распознавателя мутаций, но гораздо безопаснее верифицировать запрос на сброс пароля сначала через имейл. Ради краткости изложения мы не будем здесь реализовывать функциональность сброса пароля, но если вас интересуют примеры и ресурсы, которые для этого необходимы, можете посетить сообщество JavaScript Everywhere в чате Spectrum (https://spectrum.chat/jseverywhere).

Шифрование и токены

При описании потока аутентификации пользователя я упомянул шифрование и токены. Все это может звучать как темные мистические искусства, поэтому давайте уделим немного времени и взглянем на оба явления более подробно.

Шифрование паролей

Для эффективного шифрования паролей пользователей нам следует использовать комбинацию хеширования и соления. Хеширование — это процесс маски-

ровки строки текста путем преобразования ее в, казалось бы, случайную строку. Функции хеширования являются «односторонними», то есть после этого текст больше не может быть обращен в исходную строку. Когда пароль хеширован, простой текст этого пароля более не хранится в базе данных. Соление, в свою очередь, представляет собой процесс генерации произвольной строки из данных, которые будут использоваться в дополнение к хешированному паролю. Это гарантирует, что даже если два пароля пользователей окажутся одинаковыми, хешированная и соленая версии будут уникальны.

`bcrypt` — это популярная функция хеширования, основанная на шифре blowfish (https://ru.wikipedia.org/wiki/Blowfish), которая широко используется в ряде веб-фреймворков. В Node-разработке мы можем использовать модуль `bcrypt` (https://oreil.ly/t2Ppc) как для соления, так и для хеширования паролей.

В коде нашего приложения нужно будет использовать `bcrypt` и написать функцию для обработки соления и хэширования.

ПРИМЕРЫ СОЛЕНИЯ И ХЕШИРОВАНИЯ

Цель следующего примера чисто показательная. Мы интегрируем соление и хеширование пароля с помощью `bcrypt` в этой же главе, но несколько позже.

```
// Запрашиваем модуль
const bcrypt = require('bcrypt');

// "Стоимость" обработки соления данных, по умолчанию 10
const saltRounds = 10;

// Функция для хеширования и соления
const passwordEncrypt = async password => {
  return await bcrypt.hash(password, saltRounds)
};
```

В этом примере я мог бы передать пароль `PizzaP@rty99`, генерирующий соль `$2a$10$HF2rs.iYSvX1l5FPrX697O`, а также хешированный и соленый пароль `$2a$10$HF2rs.iYSvX1l5FPrX697O9dYF/O2kwHuKdQTdy.7oaMwVga54bWG`, являющийся солью с шифрованной строкой пароля.

Теперь при сверке пароля пользователя с хешированным и соленым паролем мы будем использовать `bcrypt`-метод `compare`:

```
// Пароль — это значение, предоставленное пользователем
// Хеш извлекается из нашей БД
const checkPassword = async (plainTextPassword, hashedPassword) => {
  // res либо true, либо false
  return await bcrypt.compare(hashedPassword, plainTextPassword)
};
```

Зашифровав пароли пользователей, мы сможем безопасно хранить их в базе данных.

JSON Web Token

С точки зрения пользователей было бы чрезвычайно неприятно вводить имя и пароль каждый раз, когда нужно обратиться к защищенной странице сайта или приложения. Вместо этого мы можем безопасно хранить ID пользователей на их устройствах в JSON Web Token (https://jwt.io). Тогда пользователь сможет с каждым запросом, совершаемым им со стороны клиента, отправлять этот токен, который сервер будет использовать для идентификации этого пользователя.

JSON Web Token (JWT) состоит из трех частей:

Header (заголовок)

Основная информация о токене и типе используемого алгоритма подписи.

Payload (полезная нагрузка)

Информация, которую мы намеренно сохранили в токене (например, имя пользователя или его ID).

Signature (подпись)

Средства для верификации токена.

Если мы рассмотрим токен, то увидим, что он состоит из произвольных символов, а каждая его часть отделена точкой: `xx-header-xx.yy-payload-yy.zz-signature- zz`.

В коде нашего приложения для генерации и проверки токенов мы можем использовать модуль `jsonwebtoken` (https://jwt.io)). Для этого мы передаем информацию, которую хотим сохранить, вместе с секретным паролем, который обычно хранится в файле .env.

```
const jwt = require('jsonwebtoken');

// Генерируем JWT, хранящий id пользователя
const generateJWT = await user => {
  return await jwt.sign({ id: user._id }, process.env.JWT_SECRET);
}

// Проверяем JWT
const validateJWT = await token => {
  return await jwt.verify(token, process.env.JWT_SECRET);
}
```

С JWT можно безопасно возвращать и сохранять ID пользователя в клиентском приложении.

JWT ИЛИ СЕССИИ?

Если вы имеете опыт работы с аутентификацией пользователей в веб-приложениях, то наверняка знакомы с сессиями пользователей. Информация сессии хранится локально, как правило, в cookie, и сверяется с хранилищем данных в памяти ((например, Redis (https://redis.io), хотя и традиционные базы данных также могут использоваться (https://oreil.ly/Ds-ba)). Ходит много споров о том, какой из способов лучше: JWT или сессии. Лично я считаю, что JWT предлагает большую гибкость, особенно при интеграции с не веб-средами вроде нативных мобильных приложений. Несмотря на то что сессии отлично работают с GraphQL, использование JWT также рекомендуется в GraphQL Foundation (https://oreil.ly/OAcJ_) и документации Apollo Server (https://oreil.ly/27iIm).

Интеграция аутентификации в API

Теперь, когда у вас уже сформировалось твердое понимание компонентов пользовательской аутентификации, мы перейдем к реализации возможности регистрироваться и авторизовываться в нашем приложении. Для этого мы обновим схемы GraphQL и Mongoose, а также напишем распознаватели мутаций `signUp` и `signIn`, которые будут генерировать токен пользователя и проверять его при каждом запросе к серверу.

Пользовательские схемы

Для начала мы обновим GraphQL-схему, добавив в нее тип `User` и изменив поле `author` типа `Note`, чтобы оно ссылалось на `User`. Для этого измените файл src/schema.js следующим образом:

```
  type Note {
   id: ID!
   content: String!
   author: User!
   createdAt: DateTime!
   updatedAt: DateTime!
  }

 type User {
   id: ID!
   username: String!
   email: String!
   avatar: String
   notes: [Note!]!
  }
```

При регистрации в нашем приложении пользователь отправляет имя, имейл-адрес и пароль. При авторизации же он будет отправлять мутацию, содержащую имя пользователя/имейл и пароль. Если мутация регистрации или авторизации

будет выполнена успешно, то API вернет токен в виде строки. Для реализации этого в нашу схему потребуется добавить две новые мутации в файл src/schema.js, каждая из которых будет возвращать `String`, которая и будет нашим JWT:

```
type Mutation {
  ...
  signUp(username: String!, email: String!, password: String!): String!
  signIn(username: String, email: String, password: String!): String!
}
```

Теперь, когда мы обновили GraphQL-схему, нужно еще обновить модели базы данных. Для этого мы создадим файл Mongoose-схемы src/models/user.js. Он будет настроен аналогично файлу модели `note` и содержать поля для имени пользователя, его имейла, пароля и аватара. Мы также потребуем, чтобы поля имени пользователя и имейла были уникальны в БД, установив `index: { unique: true }`.

Для создания модели базы данных пользователя введите в файл src/models/user.js следующее:

```
const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema(
  {
    username: {
      type: String,
      required: true,
      index: { unique: true }
    },
    email: {
      type: String,
      required: true,
      index: { unique: true }
    },
    password: {
      type: String,
      required: true
    },
    avatar: {
      type: String
    }
  },
  {
    // Присваиваем поля createdAt и updatedAt с типом Date
    timestamps: true
  }
);

const User = mongoose.model('User', UserSchema);
module.exports = User;
```

Теперь, когда у нас есть файл пользовательской модели, мы должны обновить файл src/models/index.js, чтобы эту модель экспортировать:

```
const Note = require('./note');
const User = require('./user');

const models = {
  Note,
  User
};

module.exports = models;
```

Распознаватели аутентификации

Написав схемы GraphQL и Mongoose, мы можем реализовать распознаватели, которые дадут пользователю регистрироваться и авторизовываться в нашем приложении.

Для начала мы добавим значение в переменную `JWT_SECRET`, расположенную в файле .env. Это значение должно быть строкой без пробелов. Оно будет использоваться для подписи JWT, что позволит нам проверять токены при декодировании.

```
JWT_SECRET=YourPassphrase
```

Создав эту переменную, мы можем импортировать необходимые пакеты в файл mutation.js. Мы будем использовать сторонние пакеты `bcrypt`, `jsonwebtoken`, `mongoose` и `dotenv`, а также импортируем утилиты Apollo Server `AuthenticationError` и `ForbiddenError`. Помимо этого, мы импортируем сервисную функцию `gravatar`, которую я включил в проект. Она будет генерировать URL Gravatar-изображения (https://en.gravatar.com) на основе имейла пользователя.

Введите в файле src/resolvers/mutation.js следующее:

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const {
  AuthenticationError,
  ForbiddenError
} = require('apollo-server-express');
require('dotenv').config();

const gravatar = require('../util/gravatar');
```

Теперь мы можем написать мутацию `signUp`. Она будет получать в качестве параметров имя пользователя, имейл и пароль. Мы нормализуем имейл и имя пользователя, удалив пробелы и преобразовав их в нижний регистр. Далее мы зашифруем пароль пользователя, используя модуль `bcrypt`. Помимо этого мы будем использовать вспомогательную библиотеку, чтобы генерировать URL адреса Gravatar-изображений для аватаров. После выполнения перечисленных

действий мы будем сохранять пользователя в БД и возвращать ему токен. Все это можно настроить в рамках блока `try/catch` так, чтобы распознаватель возвращал клиенту намеренно расплывчатую ошибку в случае, если в процессе регистрации возникнут проблемы.

Для выполнения всего этого напишите в файле src/resolvers/mutation.js мутацию `signUp`:

```
signUp: async (parent, { username, email, password }, { models }) => {
   // Нормализуем имейл
   email = email.trim().toLowerCase();
   // Хешируем пароль
   const hashed = await bcrypt.hash(password, 10);
   // Создаем url gravatar-изображения
   const avatar = gravatar(email);
   try {
     const user = await models.User.create({
       username,
       email,
       avatar,
       password: hashed
     });

     // Создаем и возвращаем json web token
     return jwt.sign({ id: user._id }, process.env.JWT_SECRET);
   } catch (err) {
     console.log(err);
     // Если при регистрации возникла проблема, выбрасываем ошибку
     throw new Error('Error creating account');
   }
},
```

Теперь если мы переключимся на GraphQL Playground, то сможем проверить мутацию `signUp`. Для этого мы пропишем ее со значениями имени пользователя, его имейла и пароля:

```
mutation {
  signUp(
    username: "BeeBoop",
    email: "robot@example.com",
    password: "NotARobot10010!"
  )
}
```

При запуске этой мутации сервер будет возвращать токен, аналогичный этому (рис. 7.1):

```
"data": {
   "signUp": "eyJhbGciOiJIUzI1NiIsInR5cCI6..."
 }
}
```

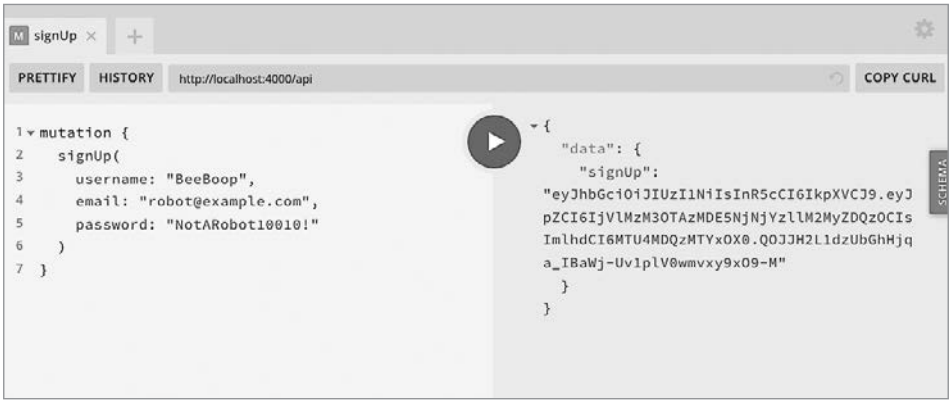

Рис. 7.1. Мутация signUp в GraphQL Playground

Следующим шагом будет написание мутации `signIn`. Она будет принимать имя пользователя, имейл и пароль. Затем будет находить пользователя в базе данных на основе его имени или имейла. После обнаружения она будет расшифровывать пароль, сохраненный в БД, и сравнивать его с введенным значением. Если эти пароли совпадут, приложение вернет пользователю токен. В противном случае будет выброшена ошибка.

Пропишите эту мутацию в файле src/resolvers/mutation.js:

```
signIn: async (parent, { username, email, password }, { models }) => {
  if (email) {
     // Нормализуем e-mail
     email = email.trim().toLowerCase();
  }

  const user = await models.User.findOne({
    $or: [{ email }, { username }]
  });

  // Если пользователь не найден, выбрасываем ошибку аутентификации
  if (!user) {
    throw new AuthenticationError('Error signing in');
  }

  // Если пароли не совпадают, выбрасываем ошибку аутентификации
  const valid = await   bcrypt.compare(password, user.password);
  if (!valid) {
    throw new AuthenticationError('Error signing in');
  }

  // Создаем и возвращаем json web token
  return jwt.sign({ id: user._id }, process.env.JWT_SECRET);
}
```

Теперь мы можем посетить GraphQL Playground и проверить мутацию `signIn`, используя учетную запись, созданную с помощью мутации `signUp`:

```
mutation {
  signIn(
    username: "BeeBoop",
    email: "robot@example.com",
    password: "NotARobot10010!"
  )
}
```

И снова в случае успеха наша мутация должна разрешаться в JWT (рис. 7.2):

```
{
  "data": {
    "signIn": "<TOKEN VALUE>"
  }
}
```

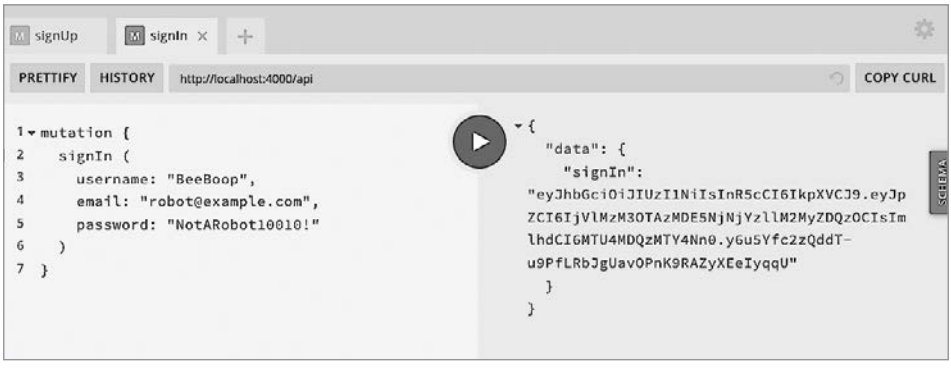

Рис. 7.2. Мутация signIn в GraphQL Playground

При наличии этих двух распознавателей пользователи смогут как регистрироваться, так и авторизовываться в нашем приложении, используя JWT. Поэкспериментируйте с добавлением других учетных записей и даже вводом неверной информации, например ошибочного пароля, чтобы увидеть возвращаемые GraphQL API результаты.

Добавление пользователя в контекст распознавателя

Теперь, когда пользователь может задействовать GraphQL-мутацию для получения уникального токена, нам придется проверять этот токен при каждом за-

просе. Следует ожидать, что наш веб-, мобильный или десктопный клиент будет отправлять токен с запросом `Authorization` в HTTP-заголовке. Затем мы сможем считывать из этого заголовка токен, декодировать его при помощи переменной `JWT_SECRET` и передавать информацию пользователя вместе с контекстом каждому GraphQL-распознавателю. С помощью этих действий можно определить, авторизованный ли пользователь делает запрос, и если это так, то какой именно.

Сначала импортируйте в файл `src/index.js` модуль `jsonwebtoken`:

```
const jwt = require('jsonвебtoken');
```

После импорта модуля можно добавить функцию, которая будет проверять действительность токена:

```
// Получаем информацию пользователя из JWT
const getUser = token => {
  if (token) {
    try {
      // Возвращаем информацию пользователя из токена
      return jwt.verify(token, process.env.JWT_SECRET);
    } catch (err) {
      // Если с токеном возникла проблема, выбрасываем ошибку
     new Error('Session invalid');
    }
  }
};
```

Теперь из заголовка каждого GraphQL-запроса мы будем захватывать токен, проверять его действительность и добавлять информацию пользователя в контекст. По завершении этого каждый GraphQL-распознаватель будет иметь доступ к ID пользователя, который мы сохранили в токене.

```
// Настраиваем Apollo Server
const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => {
    // Получаем токен пользователя из заголовков
    const token = req.headers.authorization;
    // Пытаемся извлечь пользователя с помощью токена
    const user = getUser(token);
    // Пока что будем выводить информацию о пользователе в консоль:
    console.log(user);
    // Добавляем модели БД и пользователя в контекст
    return { models, user };
  }
});
```

Несмотря на то что мы еще не выполняем пользовательское взаимодействие, можно протестировать контекст пользователя в GraphQL Playground. В левом нижнем углу UI Playground есть пространство, обозначенное как HTTP Headers.

В этой части можно добавлять заголовки, содержащие JWT, возвращенные либо в `signUp`, либо в `signIn` мутации, следующим образом (рис. 7.3):

```
{
  "Authorization": "<YOUR_JWT>"
}
```

Рис. 7.3. Заголовок авторизации в GraphQL Playground

Мы можем протестировать этот заголовок авторизации, передав его с любым запросом или мутацией в GraphQL Playground. Для этого мы напишем простой запрос `notes` и включим в него заголовок `Authorization` (рис. 7.4).

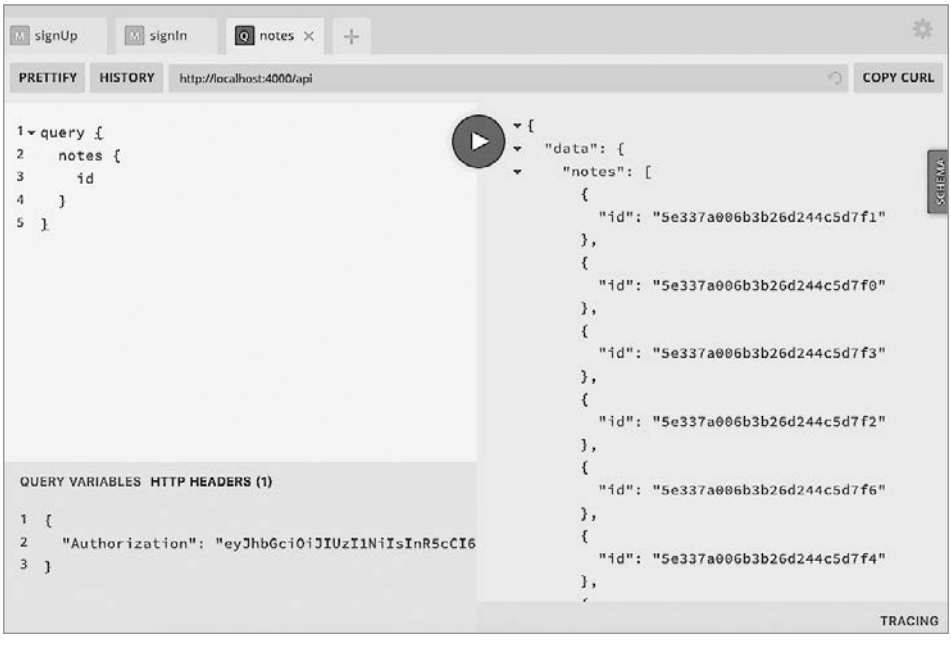

Рис. 7.4. Заголовок авторизации и запрос в GraphQL Playground

```
query {
  notes {
    id
  }
}
```

Как показано на рис. 7.5, в случае успешной аутентификации мы должны увидеть в терминале объект, содержащий ID пользователя.

Рис. 7.5. Объект пользователя, выведенный в терминал командой console.log

Теперь, разобравшись со всеми этими составляющими, мы можем аутентифицировать пользователей в нашем API.

Итоги

Потоки создания учетной записи пользователя и его авторизации могут показаться загадочными и ошеломляющими, но, разобрав их по частям, можно реализовать стабильный и безопасный процесс аутентификации в собственном API. В этой главе мы создали потоки регистрации и авторизации. Они представляют лишь небольшой фрагмент экосистемы управления учетными записями, но дают стабильный фундамент для дальнейшего построения. В следующей главе мы реализуем в нашем API пользовательские взаимодействия, которые будут присваивать право собственности на заметки и действия в рамках приложения.

ГЛАВА 8

Действия пользователя

Представьте, что вы только вступили в клуб (тот самый Секретный клуб для суперкрутых людей), но оказалось, что заняться там особо нечем. Клуб представлял собой большую пустую комнату: одни люди туда входили, другие выходили, но при этом никто из них не взаимодействовал ни с клубом, ни друг с другом. Я в некоторой степени интроверт, поэтому мне это не кажется зазорным, но платить членские взносы за такое я бы не захотел.

Пока что наш API, по существу, является большим бесполезным клубом. В нем есть возможность авторизации пользователей и сохранения их данных, но при этом у людей нет возможности владеть этими данными. Здесь мы займемся этим вопросом и добавим взаимодействие с пользователями. Мы напишем код, который превратит пользователей во владельцев собственных заметок. Они смогут ограничивать право других людей удалять или изменять записи, а также будут добавлять понравившиеся заметки в «Избранное». Кроме того, мы дадим пользователям API возможность делать вложенные запросы, позволив UI писать простые запросы, соотносящие пользователей с заметками.

Подготовка

В этой главе мы внесем в файлы заметок достаточно весомые изменения. Поскольку в БД данных у нас совсем немного, проще будет удалить существующие заметки из вашей локальной базы. Это не обязательно, но так можно уменьшить путаницу в процессе работы с этой главой.

Чтобы это сделать, перейдем в оболочку MongoDB, убедимся, что ссылаемся на базу данных `notedly` (имя БД в файле `.env`), и используем MongoDB-метод `remove()`. Наберите в терминале следующее:

```
$ mongo
$ use notedly
$ db.notes.remove({})
```

Прикрепление пользователя к новым заметкам

В предыдущей главе мы обновили файл src/index.js так, чтобы при выполнении запроса проверять наличие JWT. Если такой токен существует, мы его расшифровываем и добавляем текущего пользователя в контекст GraphQL. Это позволяет отправлять информацию о пользователе каждой функции распознавания, которую мы вызываем. Для проверки этой информации обновим существующие мутации GraphQL. Для этого используем Apollo Server-методы `AuthenticationError` и `ForbiddenError`, которые позволят нам выбрасывать ошибки, соответствующие той или иной ситуации. Это пригодится как при отладке в процессе разработки, так и для отправки корректных ответов на клиент.

Прежде чем начать, нужно импортировать пакет `mongoose` в файл распознавателей mutations.js. Это позволит правильно присваивать идентификаторы перекрестно ссылающихся MongoDB-объектов полям. Обновите импорты модулей в начале файла src/resolvers/mutation.js так:

```
const mongoose = require('mongoose');
```

Теперь в мутацию `newNote` в качестве параметра функции добавим `user`, а затем проверим, передается ли туда пользователь. Если ID не будет найден, выбросим `AuthenticationError`, поскольку не авторизированный в сервисе пользователь не может оставлять заметки. Убедившись, что запрос сделан авторизованным пользователем, можно создать в БД заметку. Теперь при этом мы будем присваивать автору ID, передаваемый в распознаватель. Это позволит ссылаться на создателя заметки непосредственно из нее.

Добавьте в src/resolvers/mutation.js следующее:

```
  // Добавляем контекст пользователя
  newNote: async (parent, args, { models, user }) => {
    // Если в контексте нет пользователя, выбрасываем AuthenticationError
    if (!user) {
      throw new AuthenticationError('You must be signed in to create a note');
    }

    return await models.Note.create({
      content: args.content,
      // Ссылаемся на mongo id автора
      author: mongoose.Types.ObjectId(user.id)
    });
  },
```

Последним шагом будет применение перекрестных ссылок на данные в БД. Для этого потребуется обновить поле `author` в схеме заметок. Сделайте это в /src/models/note.js следующим образом:

```
author: {
  type: mongoose.Schema.Types.ObjectId,
  ref: 'User',
  required: true
}
```

С этой ссылкой все новые заметки будут точно записывать автора и перекрестно ссылаться на него, исходя из контекста запроса. Давайте проверим, как это работает, написав мутацию `newNote` в GraphQL Playground:

```
mutation {
  newNote(content: "Hello! This is a user-created note") {
    id
    content
  }
}
```

При написании этой мутации мы также должны передать JWT в заголовок `Authorization` (рис. 8.1):

```
{
"Authorization": "<YOUR_JWT>"
}
```

КАК ИЗВЛЕКАТЬ JWT

Если у вас нет JWT под рукой, то вы можете выполнить мутацию `signIn` для его извлечения.

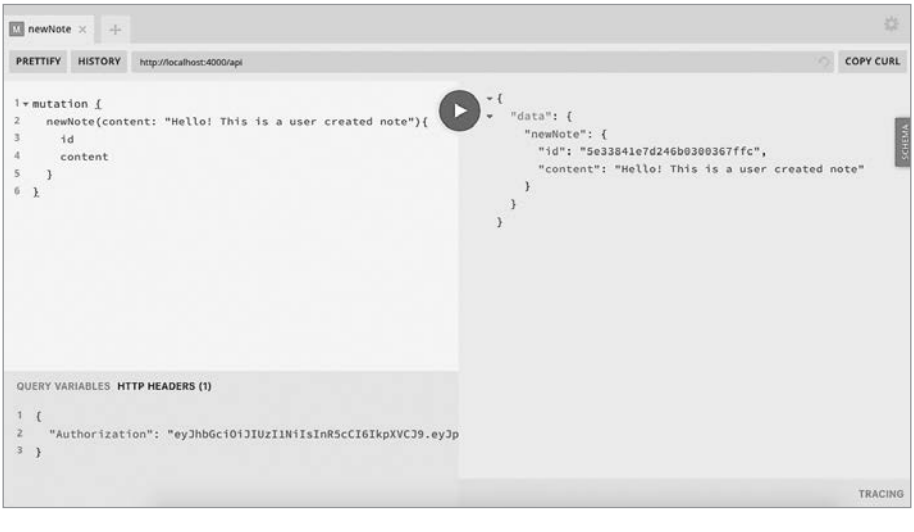

Рис. 8.1. Мутация newNote в GraphQL Playground

Пока что наш API не выдает информацию об авторе, но мы можем убедиться, что автор был добавлен правильно, просмотрев заметку в оболочке MongoDB. Наберите в терминале следующее:

```
mongo
db.notes.find({_id: ObjectId("A DOCUMENT ID HERE")})
```

Возвращенное значение должно содержать ключ автора со значением ID объекта.

Пользовательские разрешения на изменение и удаление

Теперь мы также можем добавить проверки пользователей к нашим мутациям `deleteNote` и `updateNote`. Для этого потребуется выполнять проверку передачи пользователя в контекст, а также убедиться, что именно этот пользователь — владелец данной заметки. Для этого проверим, совпадает ли ID пользователя в поле `author` нашей БД с ID, переданным в контекст распознавателя.

Обновите мутацию `deleteNote` в src/resolvers/mutation.js следующим образом:

```
deleteNote: async (parent, { id }, { models, user }) => {
  // Если не пользователь, выбрасываем ошибку авторизации
  if (!user) {
    throw new AuthenticationError('You must be signed in to delete a note');
  }

  // Находим заметку
  const note = await models.Note.findById(id);
  // Если владелец заметки и текущий пользователь не совпадают, выбрасываем
  // запрет на действие
  if (note && String(note.author) !== user.id) {
    throw new ForbiddenError("You don't have permissions to delete the note");
  }

  try {
    // Если все проверки проходят, удаляем заметку
    await note.remove();
    return true;
  } catch (err) {
    // Если в процессе возникает ошибка, возвращаем false
    return false;
  }
},
```

Теперь там же, в src/resolvers/mutation.js, обновите мутацию `updateNote` следующим образом:

```
  updateNote: async (parent, { content, id }, { models, user }) => {
    // Если не пользователь, выбрасываем ошибку авторизации
    if (!user) {
      throw new AuthenticationError('You must be signed in to update a note');
    }

    // Находим заметку
    const note = await models.Note.findById(id);
    // Если владелец заметки и текущий пользователь не совпадают, выбрасываем
    // запрет на действие
    if (note && String(note.author) !== user.id) {
      throw new ForbiddenError("You don't have permissions to update the note");
    }

    // Обновляем заметку в БД и возвращаем ее в обновленном виде
    return await models.Note.findOneAndUpdate(
      {
        _id: id
      },
      {
        $set: {
          content
        }
      },
      {
        new: true
      }
    );
  },
```

Запросы пользователей

Включив в существующую мутацию проверку пользователей, давайте также добавим в нее и пользовательские запросы. Для этого внесем три новых запроса:

`user`

На основе конкретного имени пользователя возвращает информацию о нем.

`users`

Возвращает список всех пользователей.

`me`

Возвращает информацию о текущем пользователе.

Прежде чем перейти к написанию кода распознавателя запросов, добавьте их в файл src/schema.js следующим образом:

```
type Query {
  ...
  user(username: String!):
  User users: [User!]!
  me: User!
}
```

Теперь в файле src/resolvers/query.js пропишите следующий код распознавателя запросов:

```
module.exports = {
  // ...
  // Добавляем в существующий объект module.exports следующее:
  user: async (parent, { username }, { models }) => {
    // Находим пользователя по имени
    return await models.User.findOne({ username });
  },
  users: async (parent, args, { models }) => {
    // Находим всех пользователей
    return await models.User.find({});
  },
  me: async (parent, args, { models, user }) => {
    // Находим пользователя по текущему пользовательскому контексту
    return await models.User.findById(user.id);
  }
}
```

Давайте посмотрим, как все они работают в GraphQL Playground. Для начала мы можем написать запрос user, чтобы найти информацию о конкретном пользователе. Убедитесь, что используете ранее созданное имя пользователя:

```
query {
  user(username:"adam") {
    username
    email
    id
  }
}
```

Он вернет объект данных, содержащий значение имени, имейла и ID указанного пользователя (рис. 8.2).

Теперь для поиска всех пользователей из нашей базы данных можно использовать запрос `users`, который вернет объект данных, содержащий информацию обо всех пользователях (рис. 8.3):

```
query {
  users {
    username
    email
    id
  }
}
```

Рис. 8.2. Запрос user в GraphQL Playground

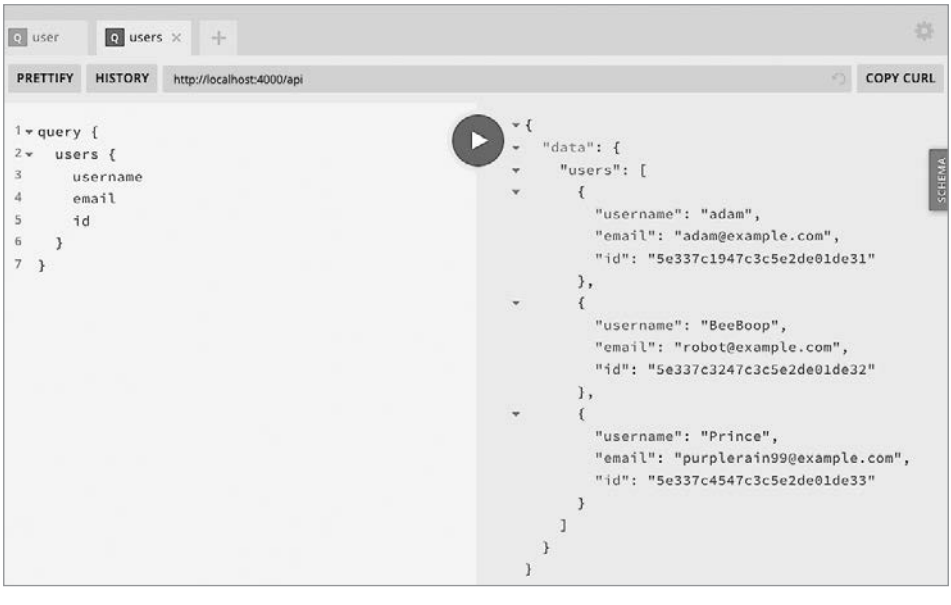

Рис. 8.3. Запрос users в GraphQL Playground

Теперь, используя запрос `me`, можно использовать JWT, переданный в HTTP-заголовке, чтобы найти информацию об авторизованном пользователе.

Для начала убедитесь, что включили токен в HTTP-заголовок в GraphQL Playground:

```
{
  "Authorization": "<YOUR_JWT>"
}
```

Теперь выполните запрос `me` (рис. 8.4):

```
query {
  me {
    username
    email
    id
  }
}
```

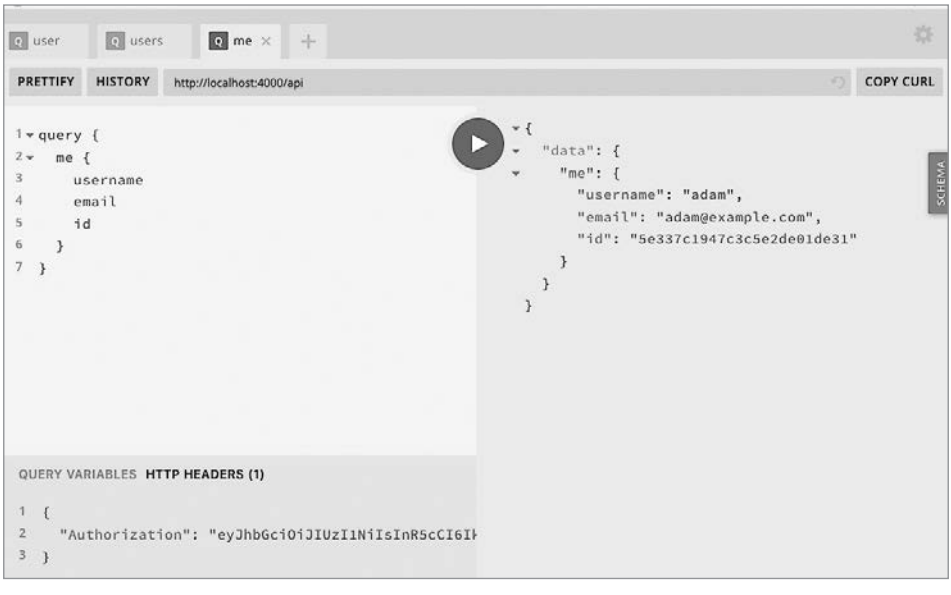

Рис. 8.4. Запрос me в GraphQL Playground

С этими распознавателями можно запрашивать из нашего API пользовательскую информацию.

Избранные заметки

Нам осталось добавить в пользовательские взаимодействия последний элемент функциональности. Как вы наверняка помните, в спецификациях нашего приложения указано, что пользователи смогут добавлять в «Избранное» чужие заметки, а также просматривать их список. Аналогично лайкам Twitter и Facebook мы хотим, чтобы наши пользователи могли сохранять заметки в «Избранном» (и удалять их оттуда). Для реализации такого поведения мы будем следовать стандартному шаблону: сначала обновим GraphQL-схему, затем модель базы данных и, наконец, функцию распознавания.

Начнем мы с изменения схемы ./src/schema.js, добавив в тип Note два новых свойства: `favoriteCount`, отслеживающее, сколько раз заметка была добавлена в избранное, и `favoritedBy`, содержащее массив пользователей, отметивших ее как избранную.

```
type Note {
  // Добавляем в тип Note следующие свойства
  favoriteCount: Int!
  favoritedBy: [User!]
}
```

Также добавим список избранных заметок в тип `User`:

```
type User {
  // Добавляем в тип User свойство favorites
  favorites: [Note!]!
}
```

Затем добавим в файл ./src/schema.js мутацию `toggleFavorite`, которая будет разрешаться либо добавлением, либо удалением отметки «Избранное» для конкретной заметки. Эта мутация будет получать в качестве параметра ID заметки и возвращать соответствующий элемент:

```
type Mutation {
  // Добавляем в тип Mutation свойство toggleFavorite
  toggleFavorite(id: ID!): Note!
}
```

Далее нам нужно обновить модель заметки для включения свойств `favoriteCount` и `favoritedBy` в БД. `favoriteCount` будет иметь тип `Number` со значением по умолчанию 0, а `favoritedBy` будет массивом объектов, содержащим ссылки на ID пользовательских объектов в базе данных. Полностью файл ./src/models/note.js теперь будет выглядеть так:

```
const noteSchema = new mongoose.Schema(
  {
    content: {
      type: String,
      required: true
    },
    author: {
      type: String,
      required: true
    },
    // Добавляем свойство favoriteCount
    favoriteCount: {
      type: Number,
      default: 0
    },
    // Добавляем свойство favoritedBy
    favoritedBy: [
```

```
      {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User'
      }
    ]
  },
  {
    // Присваиваем поля createdAt и updatedAt с типом Date
    timestamps: true
  }
);
```

Закончив обновление GraphQL-схемы и моделей БД, мы можем перейти к написанию мутации `toggleFavorite`. Эта мутация будет получать в качестве параметра ID заметки и проверять, перечислен ли данный пользователь в массиве `favoritedBy`. Если да, то будем удалять отметку «Избранное», уменьшив `favoriteCount` и удалив этого пользователя из списка. Если же пользователь еще не отмечал данную заметку как избранную — будем увеличивать `favoriteCount` на 1 и добавлять этого пользователя в массив `favoritedBy`. Для реализации описанного процесса добавьте в файл src/resolvers/mutation.js следующий код:

```
toggleFavorite: async (parent, { id }, { models, user }) => {
  // Если контекст пользователя не передан, выбрасываем ошибку
  if (!user) {
    throw new AuthenticationError();
  }

  // Проверяем, отмечал ли пользователь заметку как избранную
  let noteCheck = await models.Note.findById(id);
  const hasUser = noteCheck.favoritedBy.indexOf(user.id);

  // Если пользователь есть в списке, удаляем его оттуда и уменьшаем значение
  // favoriteCount на 1
  if (hasUser >= 0) {
    return await models.Note.findByIdAndUpdate(
      id,
      {
        $pull: {
          favoritedBy: mongoose.Types.ObjectId(user.id)
        },
        $inc: {
          favoriteCount: -1
        }
      },
      {
        // Устанавливаем new как true, чтобы вернуть обновленный документ
        new: true
      }
    );
  } else {
    // Если пользователя в списке нет, добавляем его туда и увеличиваем
    // значение favoriteCount на 1
```

```
    return await models.Note.findByIdAndUpdate(
      id,
      {
        $push: {
          favoritedBy: mongoose.Types.ObjectId(user.id)
        },
        $inc: {
          favoriteCount: 1
        }
      },
      {
        new: true
      }
    );
  }
},
```

Прописав весь этот код, пора перейти в GraphQL Playground и проверить его работу. Сделаем мы все это на примере только что созданной заметки. Начнем с написания мутации `newNote`, убедившись, что включили заголовок `Authorization` с действительным JWT (рис. 8.5):

```
mutation {
  newNote(content: "Check check it out!") {
    content
    favoriteCount
    id
  }
}
```

Вы заметите, что значение `favoriteCount` этой новой заметки автоматически выставлено как 0, потому что это значение по умолчанию, указанное нами в модели данных. Теперь давайте напишем мутацию `toggleFavorite`, чтобы отметить заметку как избранную, передав в качестве параметра ее ID. И снова необходимо убедиться, что вы включили HTTP-заголовок `Authorization` с действительным JWT.

```
mutation {
  toggleFavorite(id: "<YOUR_NOTE_ID_HERE>") {
    favoriteCount
  }
}
```

После выполнения этой мутации значение свойства `favoriteCount` заметки должно стать 1. Если выполнить мутацию повторно, значение `favoriteCount` снова уменьшится до 0 (рис. 8.6).

Теперь пользователи могут добавлять заметки в избранное и убирать их оттуда. Самое главное, что эта функциональность демонстрирует, как можно добавлять новые функции в API GraphQL-приложений.

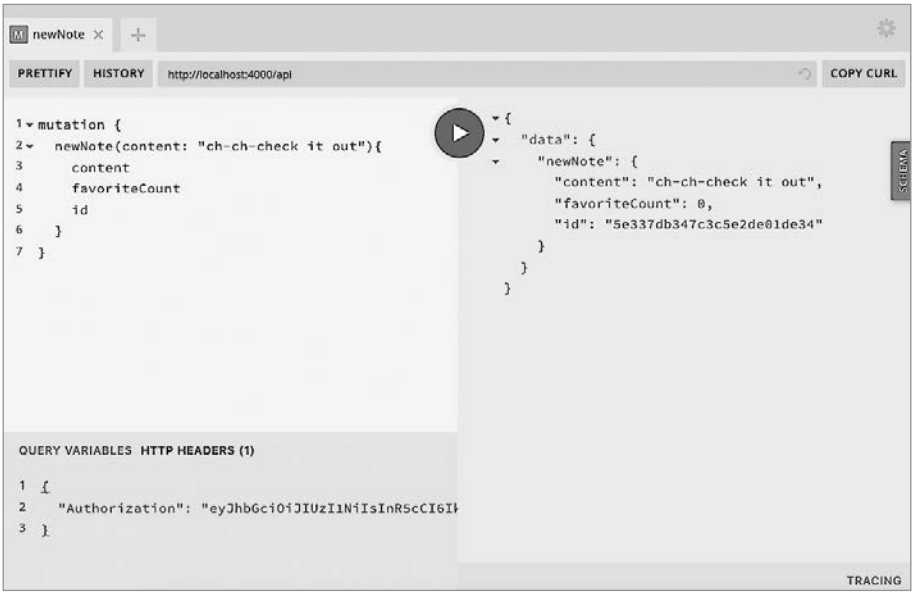

Рис. 8.5. Мутация newNote

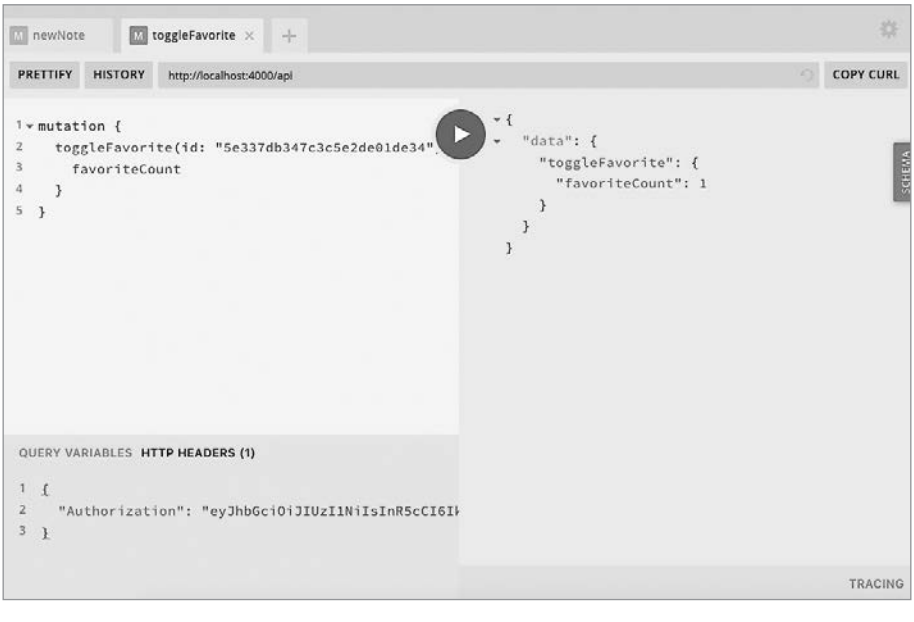

Рис. 8.6. Мутация toggleFavorite

Вложенные запросы

Одна из наиболее значимых возможностей GraphQL — это возможность вкладывать запросы, что позволяет нам писать вместо нескольких всего один запрос, возвращающий только нужные нам данные. В нашей GraphQL-схеме тип `User` включает список заметок автора, представленный в виде массива, а тип `Notes` включает ссылку на их автора. В результате мы можем извлечь список заметок из запроса `user` или получить информацию об авторе из запроса `note`.

Это означает, что можно написать подобный запрос:

```
query {
  note(id: "5c99fb88ed0ca93a517b1d8e") {
    id
    content
    # Информация об авторе заметки
    author {
      username
      id
    }
  }
}
```

Если сейчас мы попытаемся выполнить вложенный запрос наподобие предыдущего, то получим ошибку, так как мы еще не прописали код распознавателя, выполняющего поиск данной информации в БД.

Чтобы активировать эту функцию, мы добавим в директорию src/resolvers два новых файла.

Добавьте в файл src/resolvers/note.js следующее:

```
module.exports = {
  // При запросе разрешается информация об авторе заметки
  author: async (note, args, { models }) => {
    return await models.User.findById(note.author);
  },
  // При запросе разрешается информация favoritedBy для заметки
  favoritedBy: async (note, args, { models }) => {
    return await models.User.find({ _id: { $in: note.favoritedBy } });
  }
};
```

Добавьте в src/resolvers/user.js следующее:

```
module.exports = {
  // При запросе разрешается список заметок автора
  notes: async (user, args, { models }) => {
   return await models.Note.find({ author: user._id }).sort({ _id: -1 });
  },
  // При запросе разрешается список избранных заметок
```

```
  favorites: async (user, args, { models }) => {
    return await models.Note.find({ favoritedBy: user._id }).sort({ _id: -1 });
  }
};
```

Теперь нужно обновить файл src/resolvers/index.js, чтобы импортировать и экспортировать эти новые модули распознавания. В итоге он должен выглядеть так:

```
const Query = require('./query');
const Mutation = require('./mutation');
const Note = require('./note');
const User = require('./user');
const { GraphQLDateTime } = require('graphql-iso-date');

module.exports = {
  Query,
  Mutation,
  Note,
  User,
  DateTime: GraphQLDateTime
};
```

Вот теперь, если мы напишем вложенный запрос или мутацию, то получим ожидаемую информацию. Вы можете проверить это, написав следующий запрос note:

```
query {
  note(id: "<YOUR_NOTE_ID_HERE>") {
    id
    content
    # информация об авторе заметки
    author {
      username
      id
    }
  }
}
```

Этот запрос должен корректно разрешаться в выдачу имени автора и его ID. Еще одним практическим примером будет возврат информации о пользователях, сохранивших заметку в «Избранное»:

```
mutation {
  toggleFavorite(id: "<YOUR NOTE ID>") {
    favoriteCount
    favoritedBy {
      username
    }
  }
}
```

Добавив вложенные распознаватели, мы можем писать точные запросы и мутации, которые будут возвращать именно нужные нам данные.

Итоги

Поздравляю! В этой главе наш API достиг того уровня, когда пользователи могут по-настоящему с ним взаимодействовать. Он демонстрирует истинный потенциал GraphQL по интегрированию действий пользователей, добавлению новых функций и вкладыванию распознавателей. При этом мы следовали проверенному временем шаблону добавления кода в проекты: сначала прописали GraphQL-схему, затем модель БД и, наконец, код распознавателей для запроса или обновления данных. Разбив весь процесс на эти три шага, можно добавлять в приложение любые функциональные возможности. В следующей главе мы рассмотрим заключительные шаги, необходимые для подготовки нашего API к производственной среде, включая пагинацию и безопасность.

ГЛАВА 9

Детали

Когда был выпущен популярный освежитель воздуха Febreze, он потерпел неудачу. Изначально в рекламе показывали людей, использующих его для удаления конкретных неприятных запахов вроде сигаретного дыма, что привело к низкому показателю продаж. Столкнувшись с этим провалом, команда маркетологов сместила акцент на использование Febreze в качестве завершающей детали. Теперь в рекламе стали показывать человека, моющего комнату, взбивающего подушки и завершающего весь процесс уборки распылением данного средства. Такой разворот подачи продукта привел к тому, что продажи взлетели до небес.

Это отличный пример того, насколько важны могут быть детали. Пока что у нас есть рабочий API, но ему не хватает финальных штрихов, которые позволят отправить его в продакшен. В данной главе мы реализуем передовые методы обеспечения безопасности веб- и GraphQL-приложений наряду с практиками по улучшению пользовательского опыта. Эти детали, намного превосходящие по важности освежитель воздуха, окажутся решающими для безопасности, надежности и удобства использования нашего приложения.

Передовые методы Express.js для веб-приложений

Express.js — это лежащий в основе веб-приложений фреймворк, на котором работает наш API. Мы можем внести несколько небольших доработок в код нашего Express.js, чтобы обеспечить приложению прочную основу.

Express Helmet

Промежуточное ПО Express Helmet (https://oreil.ly/NGae1) является коллекцией небольших функций для обеспечения безопасности. С их помощью мы скорректируем HTTP-заголовки нашего приложения, сделав их более безопасными. Несмотря на то что многие из этих функций относятся конкретно к браузерным приложениям, добавление Helmet станет простым шагом для защиты нашего приложения от распространенных интернет-уязвимостей.

Чтобы использовать промежуточное ПО Helmet, нужно добавить его в приложение и дать команду Express задействовать его в начале общего стека всего аналогичного ПО. Добавьте в файл ./src/index.js следующее:

```
// Сначала затребуем пакет в начале файла
const helmet = require('helmet')

// Добавляем промежуточное ПО в начало стека, после const app = express()
app.use(helmet());
```

Добавляя промежуточное ПО Helmet, мы оперативно применяем в приложении передовые методы безопасности.

Совместное использование ресурсов между источниками

Совместное использование ресурсов между источниками (Cross-Origin Resource Sharing, CORS) — это средство, позволяющее выполнять запрос ресурсов из другого домена. Поскольку код нашего API и UI будет располагаться раздельно, нам понадобится включить учетные данные из других источников. Если вам интересно подробно изучить все аспекты CORS, я рекомендую ознакомиться с ним в руководстве от Mozilla (https://developer.mozilla.org/ru/docs/Web/HTTP/CORS).

Для активации CORS мы добавим в файл .src/index.js пакет его промежуточного ПО (https://oreil.ly/lYr7g):

```
// сначала затребуем пакет в начале файла
const cors = require('cors');

// добавляем промежуточное ПО после app.use(helmet());
app.use(cors());
```

Добавляя это ПО таким образом, мы разрешаем выполнение запросов разными источниками из всех доменов. Пока что это работает прекрасно, так как мы находимся в режиме разработки и, скорее всего, будем использовать домены, созданные нашими провайдерами. Но с помощью промежуточного ПО мы также можем ограничить эти запросы по их источникам.

Пагинация

Сейчас наши запросы `notes` и `users` возвращают из базы данных полный список заметок и пользователей. Это отлично работает в локальной среде разработки, но по мере роста приложения этот процесс станет неустойчивым, потому что запрос, потенциально возвращающий сотни или даже тысячи заметок, будет очень затратным и замедлит работу всей БД, сервера и сети. Вместо этого можно разбивать запросы на страницы (выполнять пагинацию), возвращая только заданное количество результатов.

Существует два распространенных типа пагинации, которые можно реализовать. Первый называется offset-пагинация: клиент передает значение смещения (offset) и возвращает ограниченное количество данных. Например, если каждая страница данных ограничена 10 записями, а мы хотим запросить третью страницу данных, то можем указать значение смещения 20. Несмотря на то что такой подход самый простой, он может привести к проблемам масштабирования и производительности.

Второй тип — это курсорная пагинация, при которой в качестве стартовой точки передается временной курсор или уникальный идентификатор. Затем запрашивается конкретное количество данных, следующих за этой записью. Такой подход дает максимальный контроль над пагинацией. Помимо этого, поскольку идентификаторы Mongo-объекта упорядочены (начинаются с 4-битного значения времени), можно легко использовать их в качестве курсора. Более подробно ознакомиться с ID Mongo-объекта вы можете в соответствующей документации по MongoDB (https://oreil.ly/GPE1c).

Если для вас все это прозвучало слишком концептуально, ничего страшного. Давайте рассмотрим реализацию разбитого на страницы списка заметок в виде GraphQL-запроса. Для начала определим, что мы будем создавать, затем обновим GraphQL-схему, а в конце и код распознавателя. Для нашего списка потребуется запросить API, при желании параллельно передавая в качестве параметра курсор. В итоге API должен вернуть ограниченное количество данных, точку курсора, представляющую последний элемент в наборе этих данных, и логическое значение в случае наличия дополнительной страницы данных для запроса.

Исходя из такого описания, можно обновить файл src/schema.js для определения этого нового запроса. Сначала нужно добавить в него тип `NoteFeed`:

```
type NoteFeed {
  notes: [Note]!
  cursor: String!
  hasNextPage: Boolean!
}
```

Затем мы добавим запрос `noteFeed`:

```
type Query {
  # Добавляем к существующим запросам noteFeed
  noteFeed(cursor: String): NoteFeed
}
```

Обновив схему, мы можем написать для этого запроса код распознавателя. В экспортируемый объект файла ./src/ resolvers/query.js добавьте следующее:

```
noteFeed: async (parent, { cursor }, { models }) => {
  // Жестко кодируем лимит в 10 элементов
  const limit = 10;
  // Устанавливаем значение false по умолчанию для hasNextPage
```

```
    let hasNextPage = false;
    // Если курсор передан не будет, то по умолчанию запрос будет пуст
    // В таком случае из БД будут извлечены последние заметки
    let cursorQuery = {};

    // Если курсор задан, запрос будет искать заметки со значением ObjectId
    // меньше этого курсора
    if (cursor) {
      cursorQuery = { _id: { $lt: cursor } };
    }

    // Находим в БД limit + 1 заметок, сортируя их от старых к новым
    let notes = await models.Note.find(cursorQuery)
      .sort({ _id: -1 })
      .limit(limit + 1);

    // Если число найденных заметок превышает limit, устанавливаем
    // hasNextPage как true и обрезаем заметки до лимита
    if (notes.length > limit) {
      hasNextPage = true;
      notes = notes.slice(0, -1);
    }

    // Новым курсором будет ID Mongo-объекта последнего элемента массива списка
    const newCursor = notes[notes.length - 1]._id;

    return {
      notes,
      cursor: newCursor,
      hasNextPage
    };
  }
```

Создав данный распознаватель, мы можем выполнить запрос `noteFeed`, который вернет максимум 10 результатов. В GraphQL Playground мы можем написать его приведенным ниже образом, чтобы получить список заметок, идентификаторы их объектов, временную метку created at, курсор и логическое значение при наличии дополнительной страницы.

```
query {
  noteFeed {
    notes {
      id
      createdAt
    }
    cursor
    hasNextPage
  }
}
```

Поскольку у нас в БД более 10 заметок, в ответ мы получаем курсор, а также `hasNextPage` со значением `true`. Используя этот курсор, мы можем запросить вторую страницу списка:

```
query {
  noteFeed(cursor: "<YOUR OBJECT ID>") {
    notes {
      id
      createdAt
    }
    cursor
    hasNextPage
  }
}
```

Далее можно повторять этот процесс для каждого курсора, чье значение `hasNextPage` будет `true`. Закончив данную реализацию, мы создали разбитый на страницы список заметок. Это позволит нашему UI запрашивать конкретный набор данных, одновременно уменьшив нагрузку на сервер и базу данных.

Ограничения данных

В дополнение к организации пагинации нам понадобится ограничить количество данных, которые можно запросить через наш API. Это предотвратит запросы, способные перегрузить сервер и БД.

В этом процессе первым простым шагом будет ограничить количество данных, которые может вернуть запрос. Два вида наших запросов — `users` и `notes` — возвращают все совпадающие данные из БД. Решить это можно, установив в запросах к базе данных метод `limit()`. Например, обновить запрос `notes` в файле ./src/resolvers/query.js следующим образом:

```
notes: async (parent, args, { models }) => {
  return await models.Note.find().limit(100);
}
```

Несмотря на то что ограничение количества данных — это хорошее начало, пока что наши запросы допускают написание без ограничения по глубине. Это означает, что с помощью одного запроса можно извлечь список заметок, информацию об авторе каждой из них, список избранных заметок каждого автора, информацию об авторе для каждой из избранных заметок и т. д. Такое количество данных слишком велико для одного запроса, а ведь его можно и продолжить! Для исключения подобных чрезмерно вложенных запросов мы можем ограничить их глубину в отношении нашего API.

Кроме этого, у нас могут быть сложные запросы, которые хоть и не чрезмерно вложены, но все равно требуют много вычислительной мощности для возврата данных. Избежать подобного вида запросов можно ограничением их сложности.

Оба вида ограничений мы можем реализовать, добавив в файл ./src/index.js пакеты `graphql-depth-limit` и `graphql-validation-complexity`.

```
// Импортируем модули в начало файла
const depthLimit = require('graphql-depth-limit');
const { createComplexityLimitRule } = require('graphql-validation-
        complexity');

// Обновляем код ApolloServer, добавив validationRules
const server = new ApolloServer({
  typeDefs,
  resolvers,
  validationRules: [depthLimit(5), createComplexityLimitRule(1000)],
  context: async ({ req }) => {
    // Получаем из заголовков токен пользователя
    const token = req.headers.authorization;
    // Пробуем извлечь пользователя с помощью токена
    const user = await getUser(token);
    // Добавляем модели БД и пользователя в контекст
    return { models, user };
  }
});
```

С помощью данных пакетов мы добавили в наш API дополнительную защиту запросов. Дополнительную информацию о защите GraphQL API от вредоносных запросов вы можете найти в замечательной статье технического директора Spectrum Макса Стойбера (Max Stoiber) по ссылке https://oreil.ly/_r5tl.

Прочие соображения

После создания данного API у вас должно сформироваться прочное понимание основ разработки с помощью GraphQL. Если вы желаете углубиться в эту тему, то лучше всего будет перейти к изучению тестирования, Apollo Engine и подписок GraphQL.

Тестирование

Я признаю, что был не прав, не добавив раздел тестирования в данную книгу. Тестирование кода важно, так как позволяет с удобством вносить в него изменения и облегчает совместный труд разработчиков. В нашей настройке GraphQL хорошо то, что распознаватели являются простыми функциями, получающими параметры и возвращающими данные, что существенно упрощает тестирование логики.

Подписки

Подписки — это невероятно мощная возможность GraphQL, предлагающая простой способ интеграции в приложение шаблона «издатель–подписчик». Это означает, что UI может подписываться и получать обновления или уведомления

при публикации данных на сервере. Это делает GraphQL-серверы идеальным решением для приложений, работающих с данными в реальном времени. Более подробную информацию о подписках вы можете найти в документации Apollo Server (https://oreil.ly/YwI5_).

Платформа Apollo GraphQL

В процессе разработки нашего API мы использовали библиотеку Apollo GraphQL. В следующих главах для взаимодействия с API мы также будем использовать клиентские библиотеки Apollo. Я выбрал именно их, потому что они являются отраслевыми стандартами и обеспечивают удобство при разработке с использованием GraphQL. Если вы перенесете свое приложение в производственную среду, то компания Apollo, обслуживающая эти библиотеки, также предлагает платформу, обеспечивающую мониторинг и инструменты для GraphQL API. Подробнее об этом вы можете узнать на сайте Apollo (https://www.apollographql.com).

Итоги

В этой главе мы добавили в наше приложение финальные штрихи. Несмотря на то что можно реализовать и множество других вариантов, на данный момент мы уже разработали устойчивый MVP (minimum viable product — «минимально жизнеспособный продукт»). В текущем его состоянии наш API уже готов к запуску, и в следующей главе мы развернем его на публичном сервере.

ГЛАВА 10

Развертывание API

Представьте, что при каждом намерении пользователя обратиться к нашему API для создания, чтения, обновления или удаления заметки нам приходилось бы встречаться с ним лично, таща за собой ноутбук. Пока что именно так и работает наш API, потому что все выполняется только на нашем конкретном компьютере. Решить эту проблему мы можем, развернув приложение на веб-сервере.

В этой главе мы выполним два шага:

1. Сначала настроим удаленную БД, к которой сможет обращаться наш API.
2. Затем развернем код API на сервере и подключим его к БД.

Выполнив эти шаги, мы сможем обращаться к API с любого подключенного к интернету компьютера, включая веб-, десктопные и мобильные интерфейсы, которые мы будем разрабатывать.

Размещение базы данных

Прежде всего мы займемся развертыванием БД, для чего воспользуемся ресурсом MongoDB Atlas. Это полностью управляемое облачное решение, поддерживаемое самой организацией Mongo. К тому же там есть бесплатная версия, которая отлично подойдет для нашего начального развертывания. Теперь давайте рассмотрим этапы этого процесса.

Для начала посетите ресурс mongodb.com/cloud/atlas и создайте аккаунт. После этого вам будет предложено создать базу данных. На этом экране вы можете управлять настройками вашей БД, но я рекомендую пока придерживаться стандартных.

Они включают:

- AWS Amazon в качестве хоста базы данных, хотя среди вариантов также доступны платформы Google Cloud и Microsoft Azure.

- Ближайший регион с опцией бесплатного уровня.
- Уровень кластера со значением по умолчанию «M0 Sandbox (общий RAM, 512MB хранилища)».
- Дополнительные настройки, которые можно оставить по умолчанию.
- Имя кластера, которое тоже можно оставить по умолчанию.

Выберите Create Cluster, после чего Mongo потребуется несколько минут, чтобы настроить базу данных (рис. 10.1).

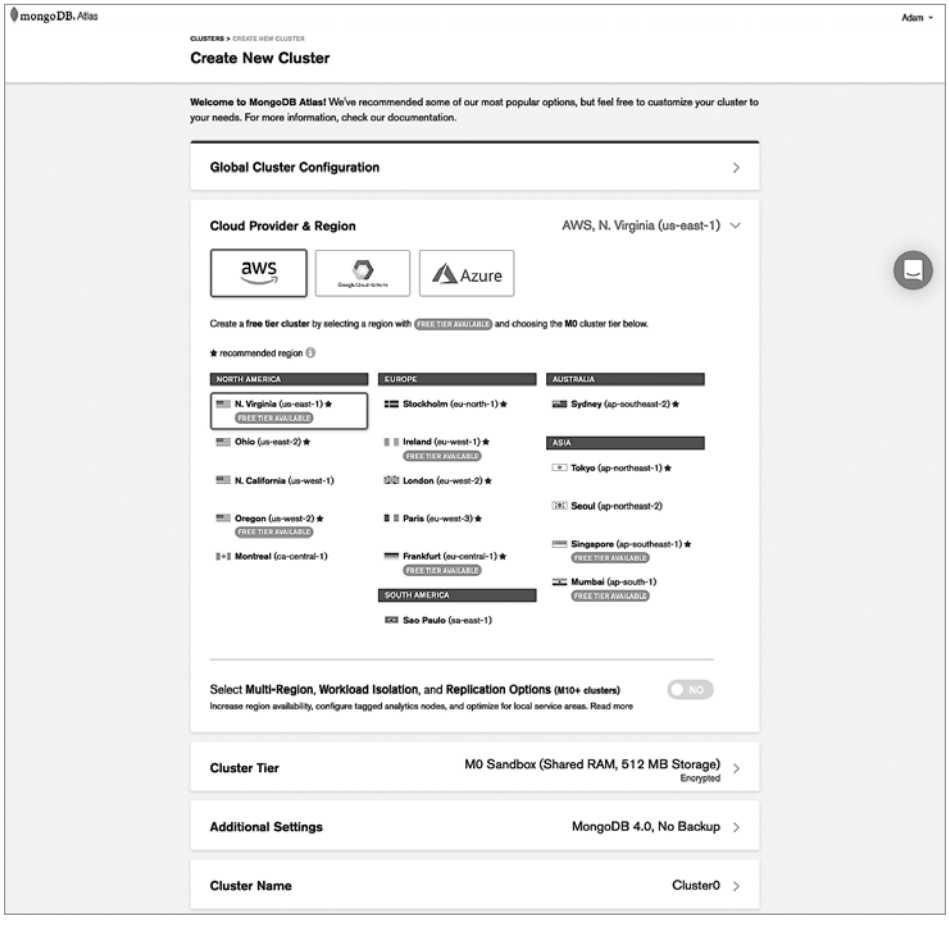

Рис. 10.1. Экран создания базы данных в MongoDB Atlas

После этого вы увидите страницу Clusters, откуда можно управлять своим отдельным кластером базы данных (рис. 10.2).

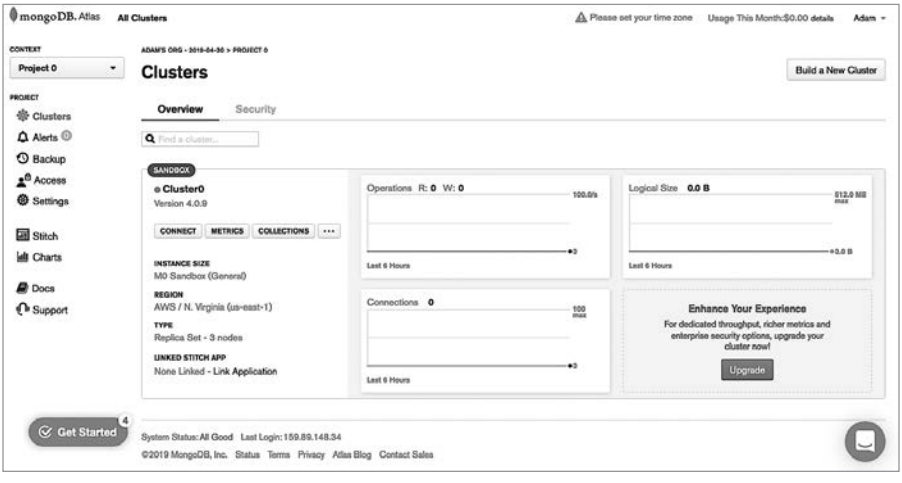

Рис. 10.2. Кластер MongoDB Atlas

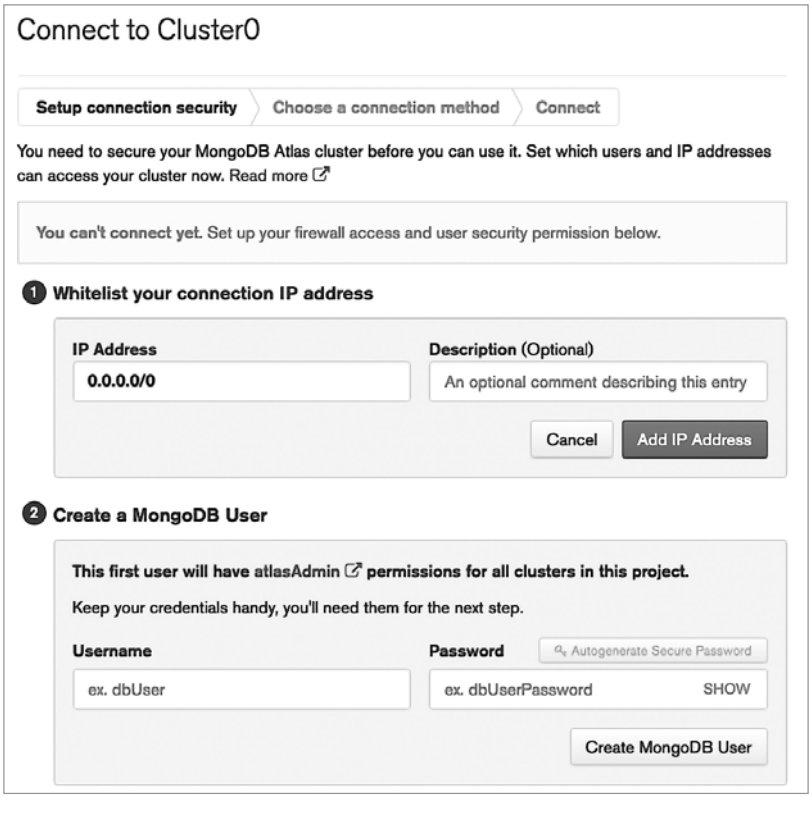

Рис. 10.3. Управление белым списком IP и аккаунтом в MongoDB Atlas

На экране Clusters кликните по Connect, где вам будет предложено настроить безопасность соединения. Первым шагом будет добавление вашего IP в белый список. Так как наше приложение будет использовать динамический IP-адрес, вам потребуется открыть этот список для любых IP-адресов, указав 0.0.0.0/0. После этого нужно будет определить надежное имя пользователя и пароль для доступа к данным (рис. 10.3).

Завершив настройку белого списка и создание аккаунта пользователя, вы перейдете к выбору метода подключения базы данных. В данном случае им будет тип подключения Application (рис. 10.4).

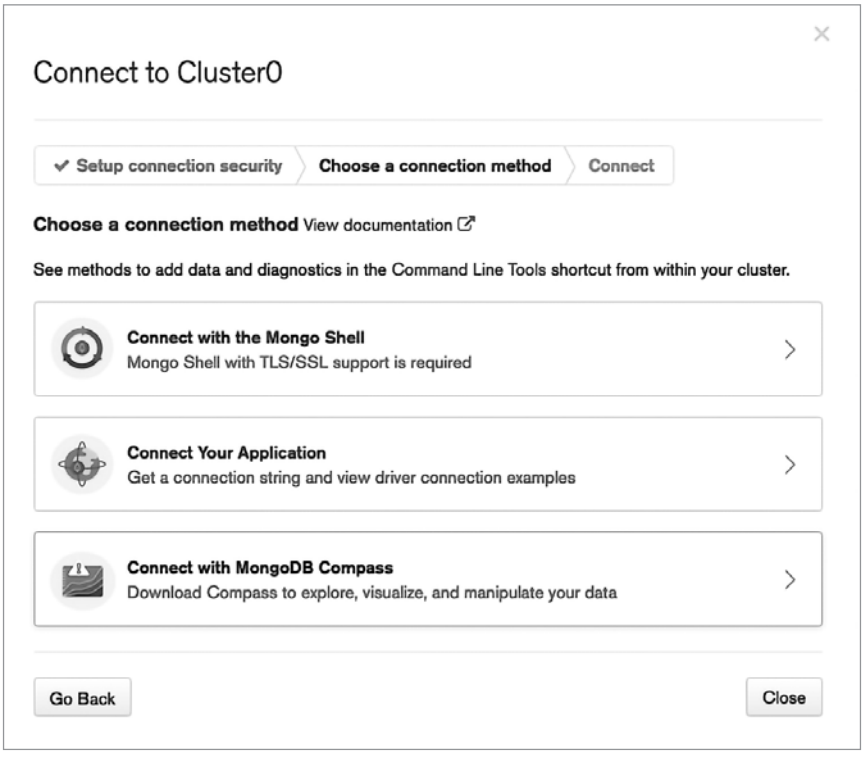

Рис. 10.4. Выбор типа подключения в MongoDB Atlas

Отсюда вы можете скопировать строку подключения, которую мы используем в продакшен-файле .env (рис. 10.5).

Настроив и запустив БД под управлением MongoDB Atlas, мы получили развернутое хранилище данных для нашего приложения. На следующем этапе мы развернем код нашего приложения и подключим его к базе данных.

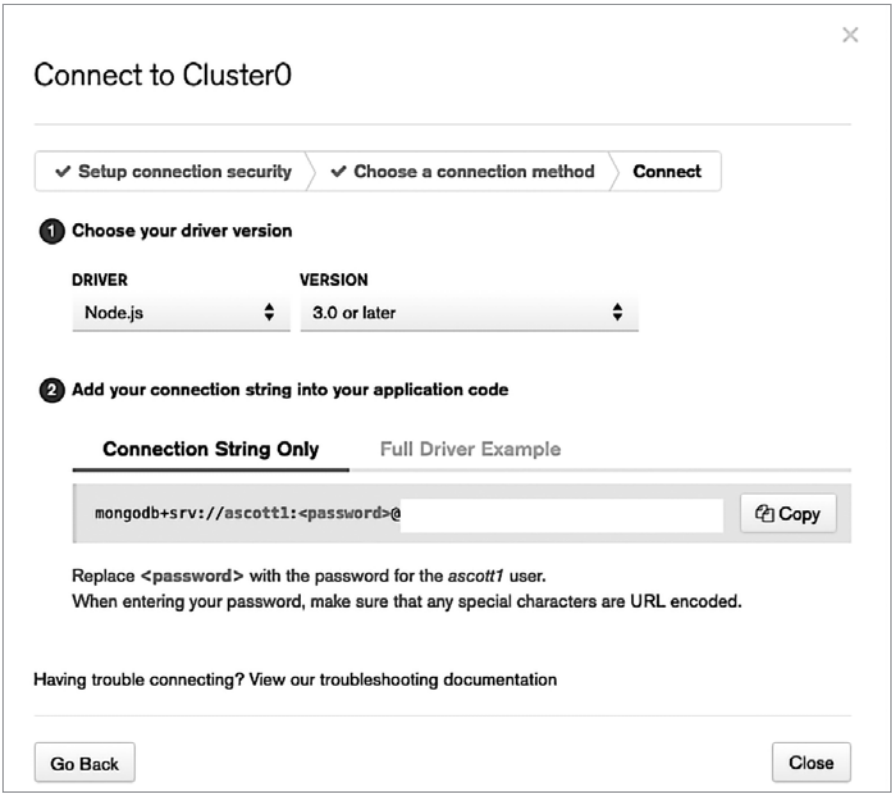

Рис. 10.5. Строка подключения базы данных MongoDB Atlas

ПАРОЛИ MONGO

В MongoDB Atlas специальные символы кодируются в шестнадцатеричном формате. Это означает, что если вы используете (а вы должны это делать!) любое небуквенное или цифровое значение, то при добавлении пароля к строке подключения для этого символа вам потребуется использовать шестнадцатеричное значение. Сайт ascii.cl предлагает соответствующие шестнадцатеричные коды для всех специальных символов. Например, если ваш пароль `Pizz@2!`, то вам потребуется закодировать символы `@` и `!`. Чтобы это сделать, нужно ввести `%`, сопровождаемый нужным шестнадцатеричным значением. В итоге ваш пароль будет выглядеть как `Pizz%402%21`.

Развертывание приложения

Далее мы займемся развертыванием кода приложения. В рамках этой книги мы будем использовать платформу облачных приложений Heroku. Я выбрал

именно ее, потому что она щедро предлагает бесплатную версию и отличается повышенным удобством в использовании. Хотя такие облачные платформы, как Google Cloud, Digital Ocean или Microsoft Azure, предоставляют альтернативные среды развертывания для приложений Node.js.

Прежде чем начать, вам потребуется посетить сайт Heroku по ссылке https://heroku.com/apps и создать аккаунт. После этого нужно будет установить для вашей ОС инструменты командной строки Heroku (https://oreil.ly/Vf2Q_).

Для macOS можно установить эти инструменты при помощи Homebrew следующим образом:

```
$ brew tap heroku/brew && brew install heroku
```

Если вы используете Windows, посетите руководство по инструментам командной строки Heroku и загрузите соответствующий установщик.

Настройка проекта

После установки инструментов командной строки мы можем настроить наш проект на сайте Heroku. Создайте новый проект, кликнув New → Create New App (рис. 10.6).

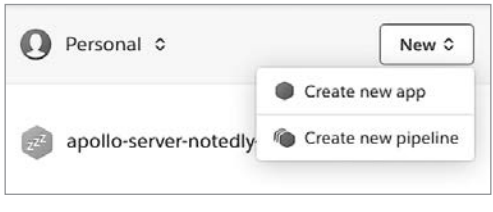

Рис. 10.6. Диалоговое окно создания нового приложения на Heroku

Далее вам будет предложено присвоить приложению уникальное имя, после чего можете нажать кнопку Create App (рис. 10.7). Забегая вперед, скажу, что данное имя нужно использовать везде, где увидите YOUR_APP_NAME.

Теперь можно добавить переменные среды. Аналогично тому, как мы использовали файл .env локально, можно управлять переменными производственной среды в рамках интерфейса сайта Heroku. Для этого перейдите на вкладку Settings, а затем нажмите кнопку Reveal Config Vars. На появившемся экране добавьте следующие переменные конфигурации (рис. 10.8):

```
NODE_ENV production
JWT_SECRET A_UNIQUE_PASSPHRASE
DB_HOST YOUR_MONGO_ATLAS_URL
```

Настроив приложение, можно перейти к развертыванию кода.

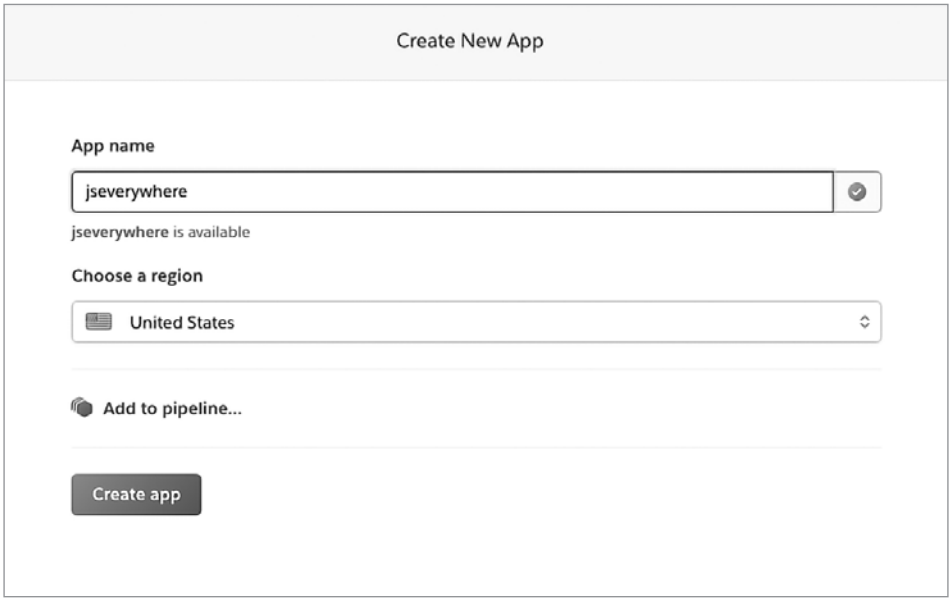

Рис. 10.7. Создание уникального имени приложения

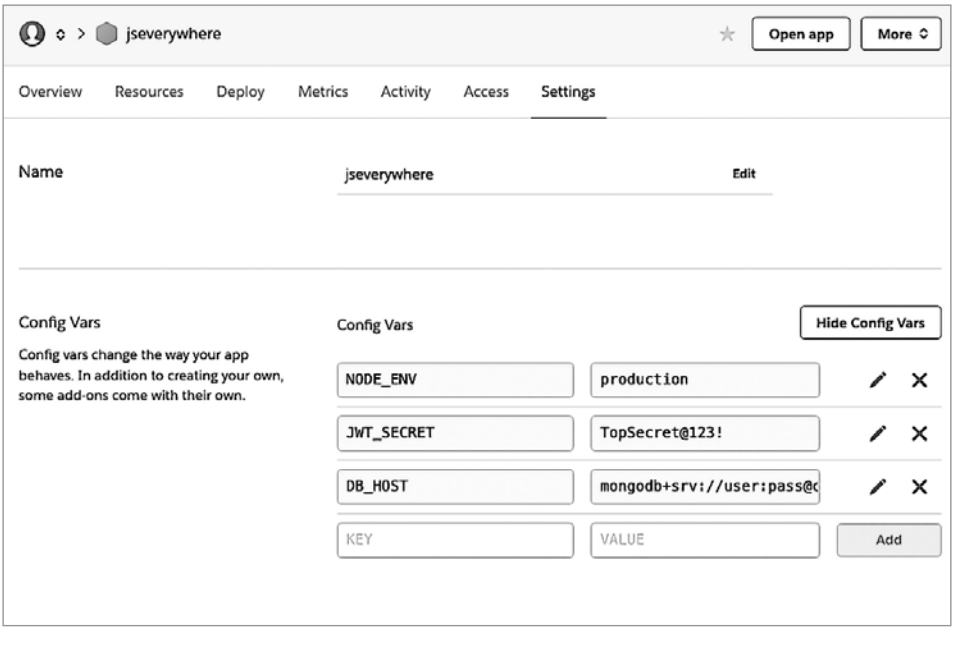

Рис. 10.8. Настройка переменных среды в Heroku

Развертывание

Теперь мы готовы к развертыванию кода на серверах Heroku. Для этого можно использовать простые команды Git из приложения терминала. Мы настроим Heroku в качестве удаленной конечной точки, затем добавим и закоммитим наши изменения и в завершение передадим код на Heroku. Чтобы это сделать, выполните в терминале следующие команды:

```
$ heroku git:remote -a <YOUR_APP_NAME>
$ git add .
$ git commit -am "application ready for production"
$ git push heroku master
```

В процессе построения и развертывания файлов на Heroku вы должны видеть вывод в терминале. Когда процесс будет завершен, для запуска нашего приложения на своих серверах Heroku будет использовать сценарий `run` в файле package.json.

Тестирование

После успешного развертывания приложения мы сможем выполнять запросы от GraphQL API к нашему удаленному серверу. По умолчанию UI GraphQL Playground в производственной среде отключен, но мы можем протестировать приложение, выполнив запрос `curl` из терминала. Для этого введите в его окне следующее:

```
$ curl \
  -X POST \
  -H "Content-Type: application/json" \
  --data '{ "query": "{ notes { id } }" }' \
  https://YOUR_APP_NAME.herokuapp.com/api
```

Если тест пройдет успешно, мы должны получить ответ, содержащий пустой массив `notes`, поскольку БД в продакшене пока что не содержит данных:

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

На этом развертывание нашего приложения закончено!

Итоги

В этой главе для развертывания базы данных и кода приложения мы использовали облачные сервисы. MongoDB Atlas и Heroku позволяют разработчикам запускать небольшие приложения и масштабировать их в любом направлении, начиная от любительских проектов и заканчивая бизнес-средами с высоким траффиком. Развернув API, мы успешно завершили разработку бэкенд-служб стека наших приложений. В следующих главах мы сфокусируемся уже на UI приложения.

ГЛАВА 11

Интерфейсы пользователей и React

В 1979-м Стив Джобс посетил Xerox Parc, где наблюдал демонстрацию персонального компьютера Xerox Alto. В то время компьютеры еще управлялись набором команд, Alto же использовал мышь и предлагал графический интерфейс из окон, которые можно было открывать и закрывать. Джобс позаимствовал эти идеи при создании оригинального Apple Macintosh. Популярность этого Mac привела к быстрому распространению компьютерных UI. Сегодня же в течение одного привычного дня мы можем взаимодействовать с десятками графических интерфейсов пользователей, среди которых могут быть ПК, смартфоны, игровые консоли, банкоматы, платежные терминалы и многое другое. UI буквально окружают нас, работая на всевозможных устройствах с любым контентом на всех размерах экранов и предлагая различные форматы взаимодействия.

В качестве примера расскажу, как я недавно ездил в другой город на встречу. Тем утром я проснулся и через телефон проверил статус своего рейса. Затем я поехал в аэропорт на своей машине, в которой экран показывал карту и позволял выбрать музыку для прослушивания. По дороге я остановился у банкомата, чтобы обналичить немного денег, ввел пин-код и проследовал инструкциям на сенсорном экране. Я прибыл в аэропорт и зарегистрировался на рейс у стойки регистрации. Я ответил на несколько имейлов на своем планшете, пока ожидал приглашения на посадку. Во время самого полета я читал книгу на устройстве с экраном e-ink. После приземления я заказал такси через приложение на телефоне и остановился на обед, выбрав свой стандартный заказ на экране дисплея. Во время встречи презентация проецировалась на экран, а многие из участников делали заметки в своих ноутбуках. Вернувшись позже вечером в отель, я просмотрел предлагаемые телешоу и фильмы, которые нашел с помощью экранного гида отеля. В итоге мой день был наполнен множеством UI и экранами различного вида, используемыми для выполнения задач, связанных с такими ключевыми элементами жизнедеятельности, как передвижение, финансы и развлечения.

В этой главе мы вкратце пройдем историю разработки пользовательского интерфейса в JavaScript. Вооружившись этими знаниями, мы перейдем к изучению основ JS-библиотеки React, которую будем использовать на протяжении оставшейся части книги.

JavaScript и UI

Изначально спроектированный в середине 90-х (как известно, аж за 10 дней (https://ru.wikipedia.org/wiki/JavaScript)) для расширения веб-интерфейсов, JavaScript представлял из себя скриптовый язык, вложенный в браузер. Это позволяло веб-дизайнерам и разработчикам добавлять на веб-страницы небольшие взаимодействия, что не было возможно при помощи одного только HTML. К несчастью, разные разработчики браузеров использовали отличающиеся реализации JavaScript, в связи с чем опираться на него было трудно. Это один из тех факторов, которые привели к распространению приложений, спроектированных для работы в одном браузере.

В середине 2000-х jQuery (а также аналогичные библиотеки вроде MooTools) начали набирать популярность. jQuery позволяла разработчикам писать код JS с помощью простого API, который работал на всех браузерах. Вскоре после этого все уже удаляли, замещали и анимировали элементы на страницах. Примерно в то же время Ajax (asynchronous JavaScript and XML — «асинхронный JavaScript и XML») позволил нам запрашивать данные с сервера и внедрять их на страницу. Комбинация этих двух технологий обеспечила экосистему для создания мощных интерактивных веб-приложений.

По мере роста сложности этих приложений параллельно росла потребность в организации и стандартном коде. К началу 2010-х доминирующие позиции в области JS-приложений заняли такие фреймворки, как Backbone, Angular и Ember. Работали они путем наложения структуры и реализации распространенных шаблонов приложений в коде фреймворка. Моделировались они зачастую по образу шаблона проектирования Model, View, Controller MVC («модель–представление–контроллер»). Каждый из фреймворков содержал предписания относительно всех слоев веб-приложений, предоставляя структурированный способ для обработки шаблонов, данных и пользовательских взаимодействий. Несмотря на все преимущества, это также означало, что для интеграции новых или нестандартных технологий могли потребоваться существенные усилия.

Тем временем десктопные приложения продолжали писать на системных языках программирования. Это вело к тому, что разработчики и их команды зачастую сталкивались с выбором в стиле или/или (приложение для Mac или для Windows, веб- или десктопное приложение и т. д.). С мобильными приложениями ситуация была аналогичной. Появление отзывчивого веб-дизайна означало, что дизайнеры и разработчики смогли создавать действительно невероятные сайты и приложения для мобильных браузеров, но выбор в пользу создания только веб-приложений закрывал им путь в магазины приложений для мобильных платформ. Приложения для Apple iOS писались на Objective C (а иногда на Swift), в то время как Android полагался на Java (не путайте с нашим другом JavaScript). Это означало, что веб, состоящий из HTML, CSS и JavaScript, был единственной по-настоящему универсальной платформой для пользовательского интерфейса.

Декларативные интерфейсы в JavaScript

В начале 2010-х разработчики Facebook столкнулись с проблемами организации и управления JS-кодом. В ответ на это инженер ПО Джордан Валке (Jordan Walke) написал React, взяв за основу PHP-библиотеку Facebook, XHP. React отличался от других популярных фреймворков тем, что фокусировался исключительно на визуализации UI. Для этого в нем использовался «декларативный» подход к программированию, то есть предоставляющий абстракцию, которая позволяет разработчикам сосредоточиться на описании необходимого состояния UI.

С появлением React и таких аналогичных библиотек, как Vue.js, мы увидели сдвиг в подходе к написанию пользовательских интерфейсов. Эти фреймворки предоставили средства для управления состоянием UI на уровне компонентов, что позволило обеспечить пользователям более плавный и легкий опыт использования приложений и в то же время повысить удобство разработки. С помощью таких инструментов, как Electron, предназначенного для построения десктопных приложений, и React Native для кроссплатформенных нативных мобильных приложений, разработчики получили возможность задействовать эти парадигмы во всех своих проектах.

Достаточно одного React

На протяжении оставшейся части книги для построения нашего UI мы будем использовать React. Наличие опыта работы с данной библиотекой для этого не обязательно, но при этом окажется не лишним, так как поможет лучше понимать ее синтаксис. Начнем мы с предварительной подготовки проекта при помощи `create-react-app` (https://oreil.ly/dMQyk) — инструмента, созданного командой React для быстрой настройки нового проекта и абстрагирования от таких его базовых инструментов сборки, как Webpack и Babel.

Находясь в терминале, перейдите в директорию проекта, выполнив `cd`, а затем введите следующие команды для создания приложения React в каталоге just-enough-react:

```
$ npx create-react-app just-enough-react
$ cd just-enough-react
```

Выполнение этих команд приведет к выводу каталога just-enough-react, содержащего всю структуру проекта, зависимости и сценарии разработки, необходимые для построения полноценного приложения. Для запуска этого приложения выполните:

```
$ npm start
```

Теперь наше приложение React будет доступно в браузере по ссылке http://localhost:3000 (рис. 11.1).

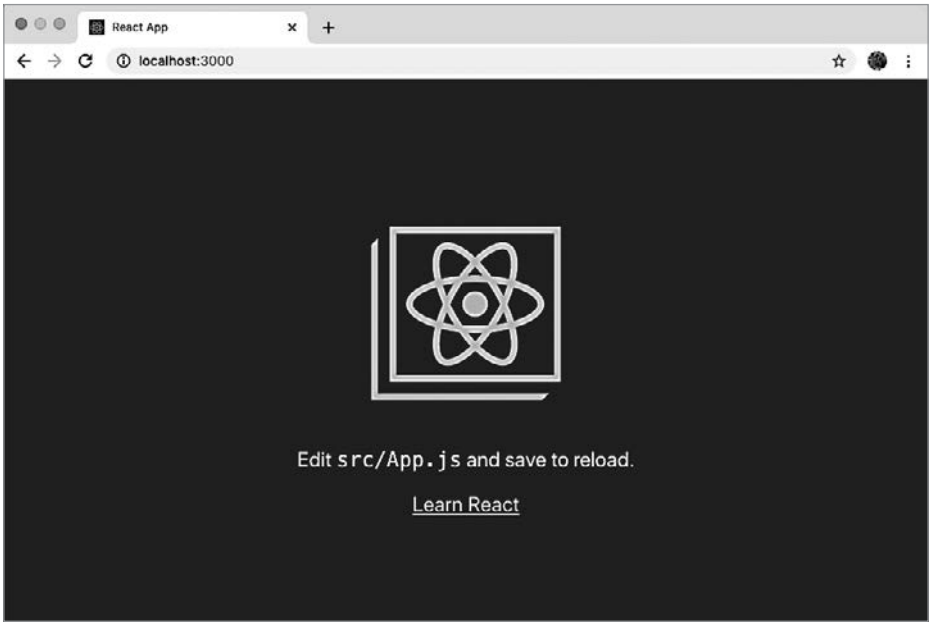

Рис. 11.1. Ввод команды npm start запустит стандартный create-react-app в браузере

Теперь можно начинать редактировать это приложение, внося изменения в файл src/App.js: он содержит наш главный компонент React. После затребования необходимых зависимостей он состоит из функции, возвращающей HTML-подобную разметку:

```
function App() {
  return (
   // Место для разметки
  )
}
```

Разметка, используемая в компоненте, называется *JSX*. JSX — это похожий на HTML синтаксис на основе XML, позволяющий нам точно описывать UI и связывать его с действиями пользователей внутри JS-файлов. Если вы знакомы с HTML, то для освоения JSX потребуется изучить лишь ряд небольших отличий. В этом примере существенное отличие в том, что HTML-свойство `class` замещено `className` для избежания коллизий с нативным JS-синтаксисом классов.

JSX? ФУ!

Если вы, как и я, имеете опыт работы с веб-стандартами и строго разделяете задачи, то это может показаться вам весьма неудобным. Я признаю, что при первом знакомстве с JSX он вызвал у меня сильное неприятие. Тем не менее связывание логики UI с выводом отображения представляет много неоспоримых преимуществ, число которых со временем может увеличиться.

Давайте перейдем к настройке нашего приложения, удалив большую часть шаблонного кода и сократив его до простого Hello World!

```
import React from 'react';
import './App.css';

function App() {
  return (
    <div className="App">
      <p>Hello world!</p>
    </div>
  );
}

export default App;
```

Вы можете заметить тег `<div>`, обертывающий все содержимое JSX. Каждый React-компонент UI должен быть включен в родительский HTML-элемент или использовать фрагмент React, представляющий контейнер для не HTML-элементов.

```
function App() {
  return (
    <React.Fragment>
      <p>Hello world!</p>
    </React.Fragment>
  );
}
```

Одна из наиболее мощных возможностей React — использование JS-кода в JSX напрямую, заключив его в фигурные скобки, `{}`. Давайте обновим функцию `App` для использования некоторых переменных:

```
function App() {
  const name = 'Adam'
  const now = String(new Date())
  return (
    <div className="App">
      <p>Hello {name}!</p>
      <p>The current time is {now}</p>
      <p>Two plus two is {2+2}</p>
    </div>
  );
}
```

В этом примере можно наглядно убедиться, что мы используем JS непосредственно в интерфейсе. Как вам такое?

Еще одна полезная особенность React заключается в способности преобразовывать каждую функциональную возможность UI в собственный компонент. В этом случае главное правило гласит, что если аспект UI работает независимо, то его нужно выделить в самостоятельный компонент. Для начала создайте новый файл src/Sparkle.js и объявите в нем функцию.

```
import React from 'react';

function Sparkle() {
  return (
    <div>

    </div>
  );
}

export default Sparkle;
```

Теперь давайте добавим функциональность. При каждом нажатии кнопки на страницу будет добавляться эмодзи sparkle («звездочка», необходимая каждому приложению функция). Для этого импортируем React-компонент `useState` и определим для него начальное состояние, которое будет пустой строкой (иначе говоря, без «звездочек»).

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

function Sparkle() {
  // Объявляем начальное состояние компонента
  // через переменную 'sparkle', являющуюся пустой строкой
  // Также определяем функцию 'addSparkle', которую
  // будем вызывать в обработчике клика
  const [sparkle, addSparkle] = useState('');

  return (
    <div>
      <p>{sparkle}</p>
    </div>
  );
}

export default Sparkle;
```

ЧТО ТАКОЕ СОСТОЯНИЕ?

В главе 15 мы рассмотрим понятие *«состояние»* более подробно. На данный же момент неплохо знать, что состояние компонента представляет текущий статус информации, которая может изменяться в рамках компонента. Например, если компонент UI имеет поле для установки флажка, то его состояние будет `true` при наличии флажка и `false` при отсутствии.

Теперь мы можем завершить наш компонент, добавив кнопку с функциональностью `onClick`. Обратите внимание на верблюжий регистр, который необходимо использовать в JSX.

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

function Sparkle() {
  // Объявляем начальное состояние компонента
  // через переменную 'sparkle', являющуюся пустой строкой
  // Также определяем функцию 'addSparkle', которую будем
  // вызывать в обработчике кликов
  const [sparkle, addSparkle] = useState('');

  return (
    <div>
      <button onClick={() => addSparkle(sparkle + '\u2728')}>
        Add some sparkle
      </button>
      <p>{sparkle}</p>
    </div>
  );
}

export default Sparkle;
```

Для использования этого компонента мы можем импортировать его в файл `src/App.js` и объявить как JSX-элемент следующим образом:

```
import React from 'react';
import './App.css';

// Импортируем компонент Sparkle
import Sparkle from './Sparkle'

function App() {
  const name = 'Adam';
  let now = String(new Date());
  return (
    <div className="App">
      <p>Hello {name}!</p>
      <p>The current time is {now}</p>
      <p>Two plus two is {2+2}</p>
      <Sparkle />
    </div>
  );
}

export default App;
```

Если теперь вы перейдете в наше приложение в браузере, то должны увидеть кнопку и иметь возможность на нее нажать, добавив таким образом на страницу «звездочку». Такова одна из супервозможностей React. Мы можем повторно

отображать конкретные компоненты или элементы компонентов отдельно от остальной части приложения (рис. 11.2).

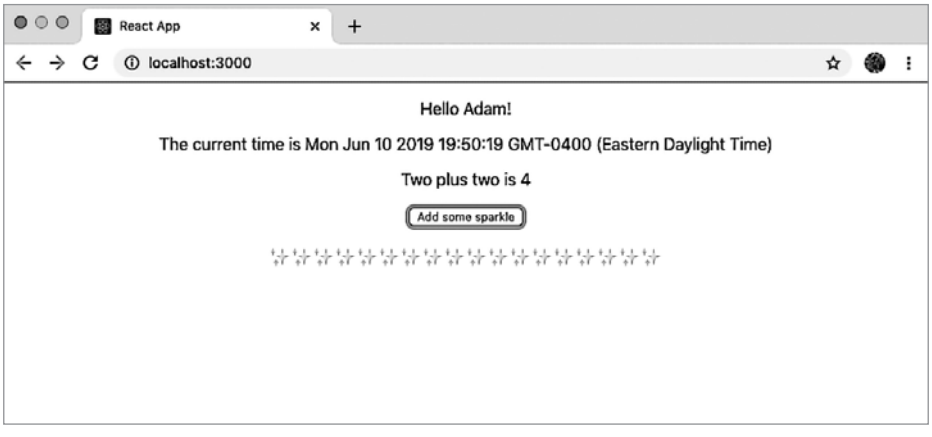

Рис. 11.2. Нажатие кнопки обновляет состояние компонента и добавляет на страницу содержимое

Мы только что создали новое приложение при помощи `create-react-app`, обновили JSX компонента Application, создали новый компонент, объявили его состояние и динамически этот компонент обновили. Теперь, имея общее представление этих основ, мы готовы переходить к разработке декларативных UI в JavaScript при помощи React.

Итоги

Мы буквально окружены пользовательскими интерфейсами, представленными во множестве разных устройств. JavaScript и веб-технологии предлагают беспрецедентные возможности разработки данных интерфейсов на многих платформах при помощи единого набора технологий. В то же время React и другие библиотеки декларативного представления позволяют нам создавать мощные динамические приложения. Комбинация этих технологий дает возможность разрабатывать удивительные продукты, не имея каких-либо особых знаний каждой платформы. В следующих главах мы применим это все на практике, использовав GraphQL API для построения интерфейсов веб-, десктопных и мобильных приложений.

ГЛАВА 12

Построение веб-клиента с помощью React

Изначальной идеей гипертекста было объединение связанных между собой документов: если научная публикация А ссылается на научную публикацию Б, то почему бы не упростить навигацию между ними, добавив возможность переходить по щелчку? В 1989-м у разработчика из CERN Тима Бернерса-Ли (Tim Berners-Lee) возникла идея совместить гипертекст с сетью компьютеров, чтобы стало возможным создавать такие связи независимо от местоположения документа. Каждое фото котика, новостная статья, твит, стрим, сайт поиска работы и обзор ресторанов в некотором смысле обязаны этой простой идее глобальной связи документов.

По своей сути, интернет остается посредником в связывании документов. Каждая страница — это HTML, отображаемый в браузере, использующий CSS для стилизации и JavaScript для улучшений и дополнений. Сегодня мы применяем эти технологии для построения всего — от личных блогов и небольших сайтов-брошюр до сложных интерактивных приложений. Основное преимущество при этом состоит в том, что интернет предоставляет универсальный доступ. Все, что вам нужно, — это браузер, установленный на подключенном к сети устройстве, по умолчанию создающий всеобъемлющую среду.

Что мы создаем

В последующих главах мы будем создавать веб-клиент для нашего приложения социальных заметок, Notedly. Пользователи получат возможность регистрироваться и заходить в свой аккаунт, писать заметки в Markdown, редактировать их, просматривать ленту заметок других пользователей и сохранять понравившиеся в «Избранное». Для реализации всего этого мы будем взаимодействовать с API GraphQL-сервера.

В нашем приложении пользователи смогут:

- создавать заметки, а также читать, редактировать и удалять их;
- просматривать ленту заметок, созданных другими, и читать их, но при этом не будут иметь возможности редактировать или удалять их;
- создавать учетную запись, авторизовываться и завершать сеанс;
- просматривать информацию своего профиля, а также публичную информацию профилей других пользователей;
- отмечать заметки других пользователей как «избранные» и просматривать их список.

Это довольно широкие возможности, но мы будем разбивать их на небольшие задачи и рассматривать поэтапно на протяжении оставшейся части книги. Освоив создание приложений React со всеми перечисленными функциями, вы сможете применять изученные инструменты и техники для построения всевозможных полнофункциональных веб-проектов.

Как мы будем это создавать

Как вы уже догадались, для создания такого приложения в качестве клиентской JS-библиотеки мы будем использовать React. Помимо этого, мы будем запрашивать данные из нашего GraphQL API. Для содействия запросам, мутациям и кэшированию данных мы дополнительно применим Apollo Client (https://oreil.ly/hAG_X). Apollo Client включает в себя коллекцию открытых инструментов для работы с GraphQL. Мы будем использовать библиотеку React, но команда Apollo также разработала интеграции Angular, Vue, Scala.js, Native iOS и Native Android.

ДРУГИЕ КЛИЕНТСКИЕ БИБЛИОТЕКИ GRAPHQL

Несмотря на то что мы будем использовать Apollo, это далеко не единственный доступный GraphQL-клиент. В качестве двух популярных альтернатив также выступают Relay от Facebook (https://relay.dev) и urql от Formiddable (https://oreil.ly/q_deu).

Кроме того, в качестве бандлера кода мы будем использовать Parcel (https://parceljs.org). Бандлер позволяет писать на JS, используя возможности, которые могут быть недоступны в браузере (например, новейшие языковые функции, модули кода, минификация), и пакетирует их для применения в среде браузера. Parcel — это не требующая настройки альтернатива таким инструментам сборки, как Webpack (https://webpack.js.org). Он предлагает много приятных возможностей вроде разделения кода и автоматического обновления браузера в процессе разработки (горячую смену модулей), но без необходимости настраивать цепочку сборки. Как вы видели в предыдущей главе, `create-react-app` (https://

oreil.ly/dMQyk) также предлагает стартовую настройку без конфигурирования, используя Webpack неявно, но Parcel позволяет создавать приложение с нуля таким способом, который я считаю идеально подходящим для обучения.

Начало

Прежде чем перейти к разработке, нам нужно сделать копию стартовых файлов проекта на свою машину. Исходный код проекта содержит все скрипты и ссылки на сторонние библиотеки, которые могут нам понадобиться для разработки приложения. Чтобы клонировать этот код на свою локальную машину, откройте терминал, перейдите в директорию, где хранятся проекты, и выполните `git clone` для репозитория проекта. Если вы также следовали процессу создания API в предыдущих главах, то у вас уже может быть создана директория notedly:

```
# Перейдите в директорию Projects
$ cd
$ cd Projects
$ # Если у вас еще нет директории notedly, введите команду `mkdir notedly`
$ cd notedly
$ git clone git@github.com:javascripteverywhere/веб.git
$ cd web
$ npm install
```

УСТАНОВКА СТОРОННИХ ЗАВИСИМОСТЕЙ

Сделав копию стартового кода книги и выполнив `npm install` в директории, вы избегаете необходимости повторно выполнять эту команду для каждой отдельной сторонней зависимости.

Структура кода следующая:

/src

Это каталог, в котором по ходу книги вы будете вести разработку.

/solutions

В этом каталоге содержатся решения для каждой главы. К нему можно обратиться при возникновении трудностей.

/final

Здесь будет содержаться итоговый рабочий проект.

Теперь, когда весь код есть на вашей локальной машине, нужно сделать копию файла .env проекта. В этом файле хранятся уникальные для нашей рабочей среды переменные. К примеру, работая локально, мы будем указывать на локальный экземпляр нашего API, но при развертывании приложения будем указывать

уже на API, развернутый удаленно. Чтобы сделать копию образца файла .env, из директории web введите в терминале следующую команду:

```
$ cp .env.example .env
```

После этого в данной директории должен появиться файл .env. Пока ничего с этим файлом делать не нужно, но по мере продвижения разработки бэкенда нашего API мы будем добавлять в него информацию. Файл `.gitignore`, содержащийся в проекте, не позволит вам по неосторожности выполнить коммит данного `.env` файла.

Я НЕ ВИЖУ .ENV-ФАЙЛ!

По умолчанию ОС скрывают файлы, начинающиеся с точки, поскольку обычно они нужны системе, а не пользователям. Если вы не видите этот файл, попробуйте открыть директорию в текстовом редакторе. В проводнике редактора он должен быть отображен. В качестве альтернативы вы можете ввести в терминале команду `ls -a`, которая выведет список всех файлов текущей рабочей директории.

Создание приложения

Скопировав стартовый код локально, мы подготовились к созданию веб-приложения на базе React. Для начала давайте заглянем в файл src/index.html. Несмотря на свою пустоту, выглядит он как стандартный HTML-файл. Но обратите внимание на следующие две строки:

```
<div id="root"></div>
<script src="./App.js"></script>
```

Для нашего приложения они будут чрезвычайно важны. При этом `root <div>` послужит для него контейнером, а App.js будет точкой входа.

Теперь можно приступать к разработке этого приложения в файле src/App.js. Если вы не пропустили введение в React в начале этой главы, то все это может показаться знакомым. В src/App.js мы начинаем с импорта библиотек `react` и `react-dom`:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

Далее мы создаем функцию `App`, которая будет возвращать содержимое приложения. На данный момент это будут всего две строчки HTML, заключенные в элемент `<div>`:

```
const App = () => {
  return (
    <div>
```

```
      <h1>Hello Notedly!</h1>
      <p>Welcome to the Notedly application</p>
    </div>
  );
};
```

ЧТО ТАКОЕ DIV?

Если вы только начинаете работать в React, то можете заинтересоваться тенденцией заключать компоненты в теги `<div>`. Компоненты React должны содержаться в родительском элементе, в качестве которого зачастую выступает тег `<div>`. При этом в его роли может также использоваться и любой другой подходящий HTML-тег, например `<section>`, `<header>` или `<nav>`. Если такой охватывающий HTML-тег кажется вам излишним, то компоненты нашего JavaScript кода можно помещать в теги `<React.Fragment>` или `<>`.

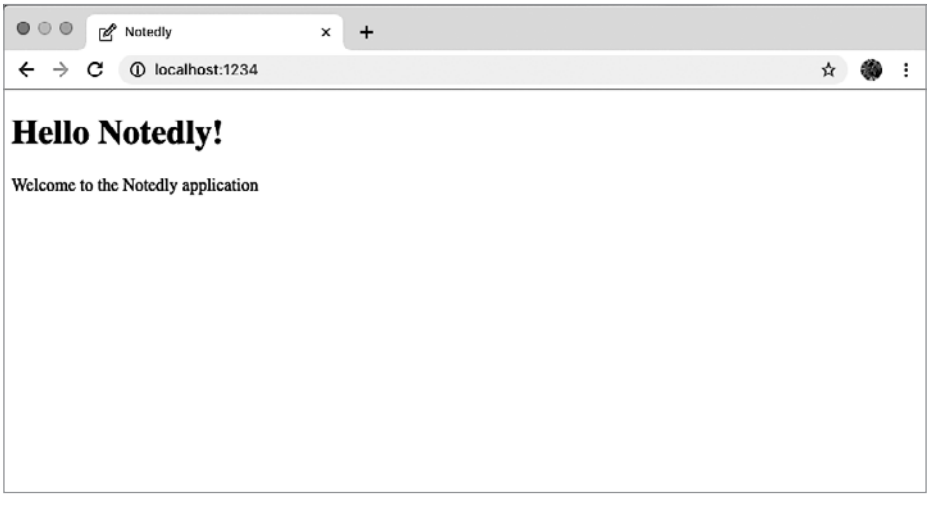

Рис. 12.1. Выполнение нашего начального приложения React в браузере

В завершение мы дадим React команду отображать приложение внутри элемента с ID `root`, добавив следующее:

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

Теперь полное содержимое файла src/App.js должно быть следующим:

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => {
  return (
```

```
    <div>
      <h1>Hello Notedly!</h1>
      <p>Welcome to the Notedly application</p>
    </div>
  );
};

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

Покончив с этим, заглянем в браузер. Запустите сервер локальной разработки с помощью команды `npm run dev` в терминале. Как только код будет собран, перейдите по ссылке http://localhost:1234, чтобы просмотреть страницу (рис. 12.1).

Маршрутизация

Одна из решающих возможностей интернета представлена возможностью связывать документы. В нашем приложении мы тоже хотим, чтобы пользователи имели возможность перемещаться между экранами или страницами. В приложениях с HTML для этого может потребоваться множество HTML-документов. При каждом переходе пользователя в новый документ этот документ будет весь повторно загружаться на двух страницах, даже если в нем есть общие аспекты вроде заголовка или футера.

В JS-приложениях можно использовать маршрутизацию на клиентской стороне. Во многом это будет аналогично созданию HTML-ссылок. Пользователь будет кликать по ссылке, вызывая обновление URL и переход на новый экран. Разница в том, что наше приложение будет обновлять только ту страницу, на которой изменилось содержимое. При этом взаимодействие будет плавным, без видимого обновления страницы.

В React для данных задач наиболее широко используется библиотека маршрутизации React Router (https://oreil.ly/MhQQR), позволяющая добавлять возможность маршрутизации в веб-приложения. Чтобы сделать это, сначала нам нужно создать директорию src/pages и включить в нее следующие файлы:

- /src/pages/index.js
- /src/pages/home.js
- /src/pages/mynotes.js
- /src/pages/favorites.js

Эти файлы, home.js, mynotes.js и favorites.js, будут отдельными компонентами страницы. Мы можем создать каждый из них с определенным начальным содержимым и хуком `effect`, который будет обновлять заголовок документа при переходе пользователя на эту страницу.

В src/pages/home.js:

```
import React from 'react';

const Home = () => {
  return (
    <div>
      <h1>Notedly</h1>
      <p>This is the home page</p>
    </div>
  );
};

export default Home;
```

В src/pages/mynotes.js:

```
import React, { useEffect } from 'react';

const MyNotes = () => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'My Notes — Notedly';
  });

  return (
    <div>
      <h1>Notedly</h1>
      <p>These are my notes</p>
    </div>
  );
};

export default MyNotes;
```

В src/pages/favorites.js:

```
import React, { useEffect } from 'react';

const Favorites = () => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Favorites — Notedly';
  });

  return (
    <div>
      <h1>Notedly</h1>
      <p>These are my favorites</p>
    </div>
  );
};

export default Favorites;
```

USEEFFECT

В предыдущих примерах для установки заголовка страницы мы использовали хук `useEffect`. Хуки эффектов позволяют добавлять сторонние эффекты в компоненты, обновляя то, что не относится к самому компоненту. Если вам интересно, можете заглянуть в документацию React для более подробного ознакомления (https://oreil.ly/VkpTZ).

Теперь при помощи пакета `react-router-dom` импортируем в src/pages/index.js React Router и методы, необходимые для выполнения маршрутизации в браузере.

```
import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';
```

Затем импортируем только что созданные компоненты страниц.

```
import Home from './home';
import MyNotes from './mynotes';
import Favorites from './favorites';
```

И в заключение назначим каждый этот компонент в качестве маршрута с конкретным URL. Обратите внимание на использование `exact` для маршрута Home, благодаря чему домашний компонент отображается только для корневого URL:

```
const Pages = () => {
  return (
    <Router>
      <Route exact path="/" component={Home} />
      <Route path="/mynotes" component={MyNotes} />
      <Route path="/favorites" component={Favorites} />
    </Router>
  );
};

export default Pages;
```

Теперь завершенный файл src/pages/index.js должен выглядеть так:

```
// Импортируем React и зависимости //маршрутизации
import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

// Импортируем маршруты
import Home from './home';
import MyNotes from './mynotes';
import Favorites from './favorites';

// Определяем маршруты
const Pages = () => {
```

```
  return (
    <Router>
      <Route exact path="/" component={Home} />
      <Route path="/mynotes" component={MyNotes} />
      <Route path="/favorites" component={Favorites} />
    </Router>
  );
};

export default Pages;
```

В завершение мы можем обновить файл src/App.js для использования маршрутов, импортировав их и отобразив компоненты.

```
import React from 'react';
import ReactDOM from 'react-dom';

// Импортируем маршруты
import Pages from '/pages';

const App = () => {
  return (
    <div>
      <Pages />
    </div>
  );
};

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

Теперь при ручном обновлении URL в браузере вы должны получить возможность просматривать каждый компонент. Введите, к примеру, **`http://localhost:1234/favorites`**, чтобы отобразить страницу «Избранное».

Создание ссылок

Мы создали страницы, но для их связи недостает ключевого компонента. Поэтому давайте добавим на нашу домашнюю страницу ссылки на другие страницы. Для этого мы используем React Router-компонент `Link`.

В src/pages/home.js:

```
import React from 'react';
// Импортируем компонент Link из react-router
import { Link } from 'react-router-dom';

const Home = () => {

  return (
    <div>
      <h1>Notedly</h1>
      <p>This is the home page</p>
```

```
      { /* добавьте список ссылок */ }
      <ul>
        <li>
          <Link to="/mynotes">My Notes</Link>
        </li>
        <li>
          <Link to="/favorites">Favorites</Link>
        </li>
      </ul>
    </div>
  );
};

export default Home;
```

Теперь у нас появится возможность перемещаться по нашему приложению. Кликая по ссылкам на домашней странице, мы будем переходить на соответствующий ее компонент. При этом такие базовые функции навигации в браузере, как кнопки «вперед» и «назад», тоже будут работать.

Компоненты UI

Мы успешно создали отдельные компоненты страницы и можем перемещаться между ними. По мере создания страниц они будут использовать несколько общих элементов пользовательского интерфейса вроде заголовка и навигации по сайту. Повторное их переписывание при каждом использовании было бы неэффективным и наверняка бы раздражало. Вместо этого мы можем написать повторно используемые компоненты UI и импортировать их в наш интерфейс там, где они нужны. В действительности представление нашего UI как состоящего из мелких компонентов — это одна из основных особенностей React, которая в свое время очень помогла мне понять и освоить данный фреймворк.

Начнем с создания компонентов заголовка и навигации. Сначала давайте создадим в директории src новый каталог components. Там мы создадим два новых файла: Header.js и Navigation.js. Компоненты React должны писаться с заглавной буквы, поэтому при именовании файлов мы также будем придерживаться такого правила.

Давайте начнем с написания компонента заголовка в src/components/Header.js. Для этого мы импортируем файл logo.svg и добавим для нашего компонента соответствующую разметку.

```
import React from 'react';
import logo from '../img/logo.svg';

const Header = () => {
  return (
    <header>
```

```
      <img src={logo} alt="Notedly Logo" height="40" />
      <h1>Notedly</h1>
    </header>
  );
};

export default Header;
```

Для компонента навигации мы импортируем React Router-функциональность `Link` и разметим неупорядоченный список ссылок. В src/components/Navigation.js:

```
import React from 'react';
import { Link } from 'react-router-dom';

const Navigation = () => {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/mynotes">My Notes</Link>
        </li>
        <li>
          <Link to="/favorites">Favorites</Link>
        </li>
      </ul>
    </nav>
  );
};

export default Navigation;
```

На скриншотах вы увидите, что я также включил символы эмодзи в качестве иконок навигации. Если вы захотите сделать то же самое, то разметка для включения эмодзи будет следующей:

```
<span aria-hidden="true" role="img">
  <!-- emoji character -->
</span>
```

Завершив написание компонентов заголовка и навигации, мы можем использовать их в приложении. Давайте обновим файл src/pages/home.js для их включения. Сначала мы их импортируем, а затем включим компонент в разметку JSX.

Теперь файл src/pages/home.js будет выглядеть так (рис. 12.2):

```
import React from 'react';

import Header from '../components/Header';
import Navigation from '../components/Navigation';
```

```
const Home = () => {
  return (
    <div>
      <Header />
      <Navigation />
      <p>This is the home page</p>
    </div>
  );
};

export default Home;
```

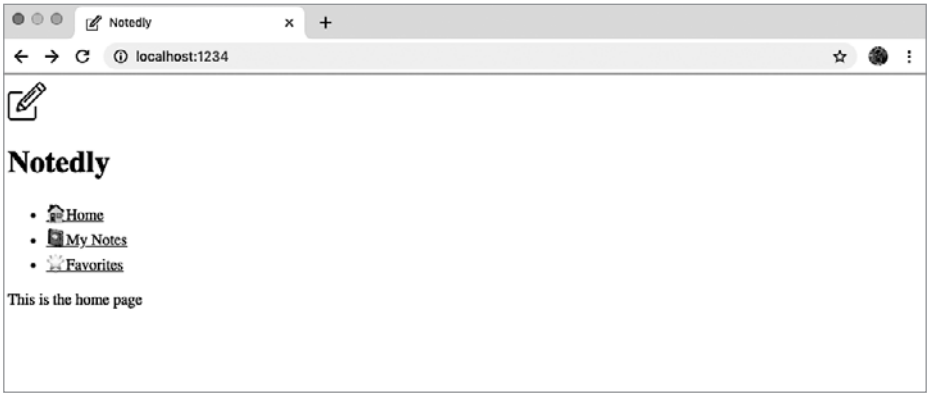

Рис. 12.2. С помощью компонентов React мы можем легко составлять общие функции UI

Это все, что нужно знать для создания в нашем приложении общих компонентов. Подробнее об использовании компонентов в UI вы можете узнать на соответствующей странице документации React «Thinking in React» (https://reactjs.org/docs/thinking-in-react.html).

Итоги

Интернет по-прежнему остается беспрецедентным посредником в распространении приложений. Он объединяет универсальный доступ с возможностью развертывания обновлений в реальном времени. В этой главе мы разработали основу JS-приложения в React. В следующей главе при помощи компонентов React и CSS-in-JS мы добавим в наше приложение макет и стиль.

ГЛАВА 13

Стилевое оформление приложения

В своей песне Lip Service 1978 года Элвис Костелло (Elvis Costello) насмешливо исполняет строчку «don't act like you're above me, just look at your shoes» («не веди себя так, будто ты выше меня, а лучше взгляни на свои ботинки»). Эта фраза подразумевает, что автор высмеивает попытку людей возвышать собственный социальный статус, просто обратив внимание на чьи-то ботинки. При этом неважно, насколько опрятен его костюм или элегантно ее платье. Хорошо это или плохо, но стиль является важным элементом человеческой культуры, и мы все привыкли отмечать подобные социальные сигналы. Археологи обнаружили, что люди даже во времена позднего палеолита создавали ожерелья и браслеты из костей, зубов, ягод и камня. Наша одежда служит не только практическим целям защиты от погодных условий, но также может сообщать окружающим о нашей культуре, социальном статусе, интересах и многом другом.

Веб-приложение вполне работоспособно и со стандартным веб-стилем, но, применив к нему CSS, мы сможем наладить более наглядную связь с нашими пользователями. В этой главе мы научимся использовать CSS-in-JS-библиотеку Styled Components, чтобы добавлять в приложение макет и стиль. Это позволит нам повысить удобство его использования и улучшить эстетический облик в рамках легко обслуживаемой структуры кода, основанной на компонентах.

Создание компонента макета

Многие, а в нашем случае все, страницы приложения будут использовать общий макет. Например, у всех страниц будет заголовок, боковая панель и область содержания (рис. 13.1). Вместо того чтобы импортировать общие элементы макета в каждый компонент, мы можем создать отдельный компонент для нашего макета и уже в него обернуть все компоненты страницы.

Начнем же мы с создания файла src/components/Layout.js. В него мы импортируем общие компоненты и содержимое макета. Наша React-функция компонента будет получать свойство `children`, которое позволит указать, где в макете бу-

дет находиться дочерний контент. Мы также используем пустой JSX-элемент `<React.Fragment>`, чтобы избежать излишней разметки.

Рис. 13.1. Структура макета страницы

Давайте создадим компонент в src/components/Layout.js:

```
import React from 'react';

import Header from './Header';
import Navigation from './Navigation';

const Layout = ({ children }) => {
  return (
    <React.Fragment>
     <Header />
      <div className="wrapper">
        <Navigation />
        <main>{children}</main>
      </div>
    </React.Fragment>
  );
};

export default Layout;
```

Теперь в файле src/pages/index.js мы можем обернуть компоненты страницы в только что созданный компонент Layout, чтобы применить общий макет к каждой странице:

```
// Импортируем React и зависимости маршрутизации
import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

// Импортируем общий компонент Layout
```

```
import Layout from '../components/Layout';

// Импортируем маршруты
import Home from './home';
import MyNotes from './mynotes';
import Favorites from './favorites';

// Определяем маршруты
const Pages = () => {
  return (
    <Router>
      {/* Оборачиваем наши маршруты в компонент Layout */}
      <Layout>
        <Route exact path="/" component={Home} />
        <Route path="/mynotes" component={MyNotes} />
        <Route path="/favorites" component={Favorites} />
      </Layout>
    </Router>
  );
};

export default Pages;
```

Заключительным шагом будет удаление всех экземпляров `<Header>` или `<Navigation>` в компонентах страницы. Например, теперь код файла src/pages/Home.js будет сокращен:

```
import React from 'react';

const Home = () => {
  return (
    <div>
      <p>This is the home page</p>
    </div>
  );
};

export default Home;
```

Закончив с этим, вы можете просмотреть приложение в браузере. При переходе между разными маршрутами на каждой странице будут появляться заголовок и ссылки навигации. Сейчас они не стилизованы, и у страницы нет визуального макета, поэтому в следующем разделе мы как раз перейдем к добавлению стилей.

CSS

Cascading Style Sheets (CSS) расшифровывается как «каскадная таблица стилей» и представляет собой набор правил, позволяющих писать стили для Сети. Каскадность стилей в данном случае означает, что отрисовывается последний или наиболее конкретно определенный стиль. Например:

```
p {
  color: green
}

p {
  color: red
}
```

Данный CSS отрисует все абзацы красным, делая правило `color: green` неактуальным. Несмотря на свою простоту, этот принцип потребовал разработки десятков паттернов и техник для избежания подводных камней. Такие структурные методы CSS, как BEM (блок–элемент–модификатор), OOCSS (объектно-ориентированный CSS) и Atomic CSS, для облегчения определения областей стилей используют описательное именование классов. Препроцессоры вроде SASS (Syntactically Awesome Style Sheets) и LeSS (Leaner Style Sheets) предоставляют инструменты, упрощающие синтаксис CSS и допускающие использование модульных файлов. Несмотря на то что у каждого из них есть свои достоинства, CSS-in-JS предлагает привлекательный способ разработки React или других приложений на основе JavaScript.

А ЧТО НАСЧЕТ CSS-ФРЕЙМВОРКОВ?

CSS- и UI-фреймворки являются популярным выбором для разработки приложений, и на то есть свои причины. Эти фреймворки выступают в качестве надежной основы стилей и уменьшают количество необходимого кода, так как предоставляют стили и функциональность для распространенных паттернов приложений. Компромисс здесь в том, что использующие их приложения могут стать визуально похожими, а также вызвать увеличение размера бандла. Однако такие компромиссы могут оказаться для вас оправданными. К числу моих любимых UI-фреймворков для работы с React относятся Ant Design (https://ant.design), Bootstrap (https://oreil.ly/XJm-B), Grommet (https://v2.grommet.io) и Rebass (https://rebassjs.org).

CSS-in-JS

При моей первой встрече с CSS-in-JS я ужаснулся. Годы моего становления в сфере веб-разработки прошли в эпоху веб-стандартов, и я продолжаю выступать за доступность и разумные усовершенствования в этой сфере. «Разделение интересов» было ключевой составляющей моих веб-практик на протяжении более 10 лет. Так что если это вам знакомо и простое прочтение CSS-in-JS вызывает у вас неприятные ассоциации, то вы не одиноки. Однако, как только я решил уделить освоению этого инструмента время, не отвлекаясь на излишнюю критику, он быстро завоевал мою симпатию. CSS-in-JS помогает легко представить интерфейс пользователя как набор компонентов, что я на протяжении многих лет старался делать с помощью структурных методов и препроцессоров CSS.

В этой книге в качестве CSS-in-JS-библиотеки мы будем использовать Styled Components (https://www.styled-components.com). Она быстра, подвижна, постоянно развивается и является наиболее популярной библиотекой для этого

инструмента. Кроме того, она используется такими крупными компаниями, как Airbnb, Reddit, Patreon, Lego, BBC News, Atlassian и др.

Styled Components позволяет нам определять стили элемента при помощи JS-синтаксиса шаблонных литералов. Мы создаем переменную, которая будет относиться к HTML-элементу и ассоциированным с ним стилям. Поскольку звучит это очень абстрактно, давайте взглянем на простой пример:

```
import React from 'react';
import styled from 'styled-components'

const AlertParagraph = styled.p`
  color: green;
`;

const ErrorParagraph = styled.p`
  color: red;
`;

const Example = () => {
  return (
    <div>
      <AlertParagraph>This is green.</AlertParagraph>
      <ErrorParagraph>This is red.</ErrorParagraph>
    </div>
  );
};

export default Example;
```

Как видите, мы можем легко определять области видимости стилей. Помимо этого, мы ограничиваем эти области до конкретного компонента. Это помогает нам избежать коллизий имен классов между разными частями приложения.

Создание компонента Button

Теперь, когда мы разобрались с основами стилизованных компонентов, давайте интегрируем их в наше приложение. Начнем же мы с написания стилей для элемента `<button>`, что позволит нам использовать его в этом приложении повторно. В предыдущем примере мы интегрировали стили вместе с кодом React/JSX, но можно также написать и автономные стилизованные компоненты. Для начала создайте новый файл src/components/Button.js, импортируйте библиотеку `styled` из `styled-components` и настройте экспортируемые компоненты в виде шаблонного литерала следующим образом:

```
import styled from 'styled-components';

const Button = styled.button`
/* our styles will go here */
`;

export default Button;
```

Создав компонент, можно заполнить его стилями. Добавьте базовые стили кнопки, а также стили hover (наведения) и active (активного состояния):

```
import styled from 'styled-components';

const Button = styled.button`
  display: block;
  padding: 10px;
  border: none;
  border-radius: 5px;
  font-size: 18px;
  color: #fff;
  background-color: #0077cc;
  cursor: pointer;

  :hover {
    opacity: 0.8;
  }

  :active {
    background-color: #005fa3;
  }
`;

export default Button;
```

Теперь мы можем использовать эту кнопку в любой части приложения. Например, для ее использования на домашней странице можно импортировать компонент и добавить элемент `<Button>` в любом месте, где мы обычно использовали бы `<button>`.

В src/pages/home.js:

```
import React from 'react';

import Button from '../components/Button';

const Home = () => {
  return (
    <div>
      <p>This is the home page</p>
      <Button>Click me!</Button>
    </div>
  );
};
export default Home;
```

Таким образом, мы написали стилизованный компонент, который можно использовать в любой части приложения. Это существенно повышает обслуживаемость, поскольку мы можем легко находить и изменять стили в базе кода. Кроме того, мы можем совмещать стилизованные компоненты с разметкой,

что позволит создавать небольшие, повторно используемые и обслуживаемые компоненты.

Добавление глобальных стилей

Несмотря на то что многие из наших стилей будут содержаться в отдельных компонентах, каждый сайт или приложение также имеет набор глобальных стилей (таких как CSS-сбросы, шрифты и базовые цвета). Для размещения этих стилей мы можем создать компонент GlobalStyle.js.

Он будет немного отличаться от нашего предыдущего примера, поскольку мы создадим таблицу стилей, а не стили, прикрепленные к конкретному HTML-элементу. Для реализации этого мы импортируем модуль `createGlobalStyle` из `styled-components`. Мы также импортируем библиотеку `normalize.css` (https://oreil.ly/i4lyd), чтобы обеспечить согласованную отрисовку HTML-элементов в браузерах. Наконец, мы добавим несколько глобальных правил для HTML `body` приложения и стилей ссылок по умолчанию.

В src/components/GlobalStyle.js:

```
// Импортируем createGlobalStyle и нормализуем
import { createGlobalStyle } from 'styled-components';
import normalize from 'normalize.css';

// Можно написать CSS как шаблонный литерал JS
export default createGlobalStyle`
  ${normalize}

  *, *:before, *:after {
    box-sizing: border-box;
  }

  body,
  html {
    height: 100%;
    margin: 0;
  }

  body {
    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto,
      Oxygen-Sans, Ubuntu, Cantarell, 'Helvetica Neue', sans-serif;
    background-color: #fff;
    line-height: 1.4;
  }

  a:link,
  a:visited {
    color: #0077cc;
  }
```

```
  a:hover,
  a:focus {
    color: #004499;
  }

  code,
  pre {
    max-width: 100%;
  }
`;
```

Для применения этих стилей мы импортируем их в файл App.js и добавим в приложение элемент `<Global Style />`.

```
import React from 'react';
import ReactDOM from 'react-dom';

// Импортируем глобальные стили
import GlobalStyle from '/components/GlobalStyle';
// Импортируем маршруты
import Pages from '/pages';

const App = () => {
  return (
    <div>
      <GlobalStyle />
      <Pages />
    </div>
  );
};

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

Теперь к приложению применены глобальные стили. При его просмотре в браузере вы увидите, что изменился шрифт, ссылки получили новые стили, а поля были удалены (рис. 13.2).

Рис. 13.2. Приложение после применения глобальных стилей

Стили компонентов

После применения глобальных стилей мы можем перейти к стилизации отдельных компонентов. При этом мы также представим общий макет нашего приложения. Для каждого стилизуемого нами компонента мы сначала импортируем библиотеку `styled` из `styled-components`. Затем в виде переменных мы определим несколько стилей элементов, а в завершение используем эти элементы в JSX нашего компонента React.

ИМЕНОВАНИЕ СТИЛИЗОВАННЫХ КОМПОНЕНТОВ

Чтобы избежать коллизий с HTML-элементами, важно именовать стилизованные компоненты с заглавных букв.

Можно начать с src/components/Layouts.js, где для макета приложения мы добавим стиль в структурные теги `<div>` и `<main>`:

```
import React from 'react';
import styled from 'styled-components';

import Header from './Header';
import Navigation from './Navigation';

// Стили компонента
const Wrapper = styled.div`
  /* Можно применить в стилизованном компоненте стили медиазапросов */
  /* Таким образом, макет будет применяться только для экранов
     шириной 700 пикселей и больше */
  @media (min-width: 700px) {
    display: flex;
    top: 64px;
    position: relative;
    height: calc(100% - 64px);
    width: 100%;
    flex: auto;
    flex-direction: column;
  }
`;

const Main = styled.main`
  position: fixed;
  height: calc(100% - 185px);
  width: 100%;
  padding: 1em;
  overflow-y: scroll;
  /* Снова применяем стили медиазапросов к экранам от 700 пикселей */
  @media (min-width: 700px) {
    flex: 1;
```

```
    margin-left: 220px;
    height: calc(100% - 64px);
    width: calc(100% - 220px);
  }
`;

const Layout = ({ children }) => {
  return (
    <React.Fragment>
      <Header />
      <Wrapper>
        <Navigation />
        <Main>{children}</Main>
      </Wrapper>
    </React.Fragment>
  );
};

export default Layout;
```

Закончив с компонентом Layout.js, мы можем добавить несколько стилей в файлы Header.js и Navigation.js.

В src/components/Header.js:

```
import React from 'react';
import styled from 'styled-components';
import logo from '../img/logo.svg';

const HeaderBar = styled.header`
  width: 100%;
  padding: 0.5em 1em;
  display: flex;
  height: 64px;
  position: fixed;
  align-items: center;
  background-color: #fff;
  box-shadow: 0 0 5px 0 rgba(0, 0, 0, 0.25);
  z-index: 1;
`;

const LogoText = styled.h1`
  margin: 0;
  padding: 0;
  display: inline;
`;

const Header = () => {
  return (
    <HeaderBar>
      <img src={logo} alt="Notedly Logo" height="40" />
      <LogoText>Notedly</LogoText>
```

```
    </HeaderBar>
  );
};

export default Header;
```

И наконец, в src/components/Navigation.js:

```
import React from 'react';
import { Link } from 'react-router-dom';
import styled from 'styled-components';

const Nav = styled.nav`
  padding: 1em;
  background: #f5f4f0;

  @media (max-width: 700px) {
    padding-top: 64px;
  }

  @media (min-width: 700px) {
    position: fixed;
    width: 220px;
    height: calc(100% - 64px);
    overflow-y: scroll;
  }
`;

const NavList = styled.ul`
  margin: 0;
  padding: 0;
  list-style: none;
  line-height: 2;

  /* Мы можем вложить стили в styled-components */
  /* Следующие стили будут применены к ссылкам в компоненте NavList */
   a {
    text-decoration: none;
    font-weight: bold;
    font-size: 1.1em;
    color: #333;
  }

  a:visited {
    color: #333;
  }

  a:hover,
  a:focus {
    color: #0077cc;
  }
`;
```

```
const Navigation = () => {
  return (
    <Nav>
      <NavList>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/mynotes">My Notes</Link>
        </li>
        <li>
          <Link to="/favorites">Favorites</Link>
        </li>
      </NavList>
    </Nav>
  );
};

export default Navigation;
```

Теперь, после применения этих стилей, наше приложение полностью оформлено (рис. 13.3). Забегая вперед, скажу, что мы можем применять стили по ходу создания отдельных компонентов.

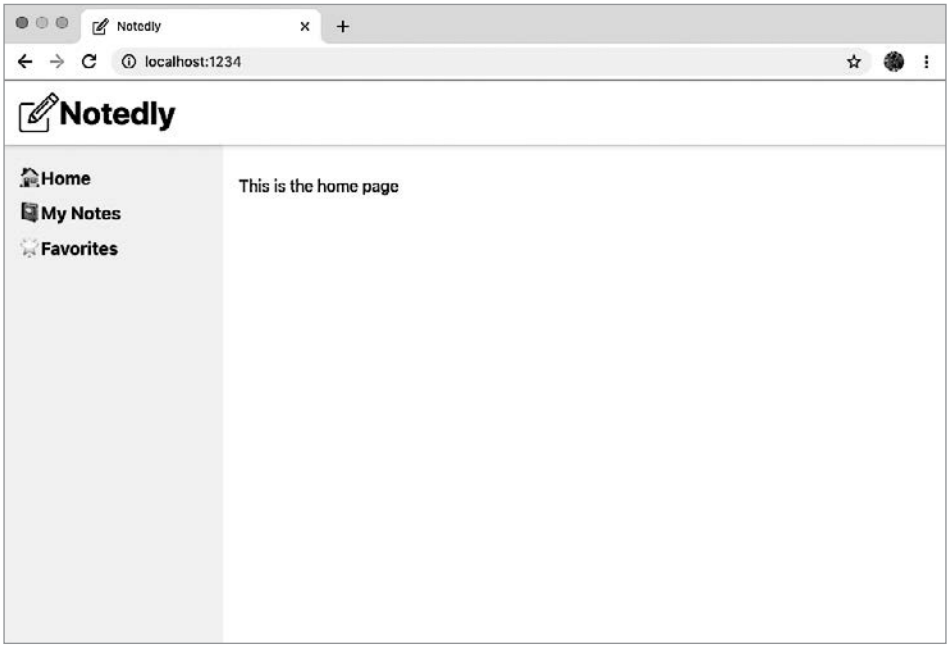

Рис. 13.3. Приложение после применения стилей

Итоги

В пройденной главе мы добавили в приложение макет и стили. Использование CSS-in-JS-библиотеки Styled Components позволяет нам писать сжатые CSS-стили, а также правильно распределять их по областям видимости. Затем эти стили можно применять к отдельным компонентам или глобально по всему приложению. В следующей главе мы будем работать над добавлением в приложение полноценной функциональности, реализуя GraphQL-клиент и делая вызовы к нашему API.

ГЛАВА 14

Работа с Apollo Client

Я отчетливо помню свое первое подключение к интернету. Модем компьютера набирал локальный номер, подключенный к провайдеру интернет-услуг (ISP), что давало мне свободу пребывания в Сети. Хотя в свое время это и казалось чем-то магическим, данная технология была очень далека от мгновенных, постоянно активных подключений, используемых нами сегодня. Вот как выглядел этот процесс тогда:

1. Сидя за компьютером, я открывал ПО ISP.
2. Нажимал Connect и ожидал, пока модем наберет номер.
3. В случае успешного подключения модем издавал победоносный звук. В случае же провала, например во время повышенной нагрузки на линии, попытка повторялась.
4. Подключившись, я получал уведомление об успехе и переходил к просмотру интернет-страниц во всей их GIF-красе 90-х.

Этот цикл может показаться непростым, но именно таким способом службы общаются друг с другом: они запрашивают соединение, выполняют это соединение, отправляют запрос и получают на него ответ. Наши клиентские приложения будут работать таким же образом. Сначала мы будем устанавливать соединение с нашим серверным приложением API и в случае успеха отправлять к этому серверу запрос.

ЛОКАЛЬНОЕ ВЫПОЛНЕНИЕ API

Разработка клиентского веб-приложения потребует доступа к локальному экземпляру нашего API. Если вы следовали процессу построения на протяжении книги, то у вас на компьютере уже должен быть работоспособный API Notedly с собственной базой данных. Если же нет, то в приложение A я добавил инструкции о том, как сделать копию рабочего API и запустить его, вместе с некоторыми примерами данных. Если у вас уже есть API, но вам нужны дополнительные данные для работы, выполните `npm run seed` из корня директории проекта API.

В этой главе для подключения к API мы будем использовать Apollo Client. После подключения мы напишем GraphQL-запрос для отображения данных на странице. Кроме этого, мы добавим пагинацию в запрос API и в компоненты интерфейса.

Настройка Apollo Client

Во многом аналогично Apollo Server, Apollo Client предлагает ряд полезных функций, упрощающих работу с GraphQL в рамках JS-приложений UI. Apollo Client предоставляет библиотеки для подключения веб-клиента к API, локального кэширования, GraphQL-синтаксиса, управления локальным состоянием и для многого другого. Мы будем также использовать Apollo Client с приложением React, но Apollo помимо этого предлагает библиотеки для Vue, Angular, Meteor, Ember и Web Components.

Сначала нам нужно убедиться, что файл .env содержит ссылку на локальный API URI. Это позволит использовать в разработке локальный экземпляр API, и в то же время будет указывать на API нашего продукта, когда мы произведем его релиз на публичном сервере. В .env-файл нужно добавить переменную `API_URI` с адресом сервера локального API.

```
API_URI=http://localhost:4000/api
```

Используемый нами бандлер кода, Parcel, настроен на автоматическую работу с файлами .env. Каждый раз, когда нам нужно сослаться на переменную .env в коде, мы можем задействовать `process.env.VARIABLE_NAME`. Это позволит использовать уникальные значения в локальной разработке, производственной и любой другой среде, которая нам может понадобиться (например, при поэтапной или непрерывной интеграции).

Сохранив адрес в переменной среды, мы готовы перейти к подключению нашего веб-клиента к серверу API. Работая в файле src/App.js, мы сначала имопортируем необходимые пакеты Apollo.

```
// Импортируем библиотеки Apollo Client
import { ApolloClient, ApolloProvider, InMemoryCache } from '@apollo/client';
```

С помощью этих данных мы можем настроить экземпляр Apollo Client, передав ему API URI, инициировать кэш и активировать использование локальных инструментов разработчика Apollo.

```
// Настраиваем API URI и кэш
const uri = process.env.API_URI;
const cache = new InMemoryCache();

// Настраиваем Apollo Client
const client = new ApolloClient({
```

```
  uri,
  cache,
  connectToDevTools: true
});
```

Наконец, мы можем подключить наше приложение React к Apollo Client, обернув его в ApolloProvider. Мы заменим пустые теги `<div>` на `<ApolloProvider>` и добавим клиент в качестве подключения.

```
const App = () => {
  return (
    <ApolloProvider client={client}>
      <GlobalStyle />
      <Pages />
    </ApolloProvider>
  );
};
```

В целом файл src/App.js теперь будет выглядеть так:

```
import React from 'react';
import ReactDOM from 'react-dom';

// Импортируем библиотеки Apollo Client
import { ApolloClient, ApolloProvider, InMemoryCache } from '@apollo/client';

// Глобальные стили
import GlobalStyle from '/components/GlobalStyle';
// Импортируем маршруты
import Pages from '/pages';

// Настраиваем API URI и кэш
const uri = process.env.API_URI;
const cache = new InMemoryCache();

// Настраиваем Apollo Client
const client = new ApolloClient({
  uri,
  cache,
  connectToDevTools: true
});

const App = () => (
  <ApolloProvider client={client}>
    <GlobalStyle />
    <Pages />
  </ApolloProvider>
);

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

Подключив клиент к серверу API, мы можем интегрировать GraphQL-запросы и мутации в приложение.

Запросы к API

При обращении к API мы запрашиваем данные. В клиенте UI нам нужна возможность запрашивать эти данные и отображать их пользователю. Apollo позволяет нам составлять запросы для получения данных. После этого можно обновить компоненты React для отображения этих данных конечному пользователю. Просмотреть использование запросов можно, написав запрос `noteFeed`, который выдаст пользователю ленту последних заметок и отобразит ее на домашней странице приложения.

Когда я впервые пишу запрос, то обычно иду таким путем:

1. Учитываю, какие данные должен вернуть запрос.
2. Пишу запрос в GraphQL Playground.
3. Интегрирую этот запрос в клиентское приложение.

Давайте составим наш запрос согласно этому плану. Если вы следовали указаниям в разделе книги, посвященном API, то можете вспомнить, что запрос `noteFeed` возвращает список из 10 заметок вместе с `cursor`, указывающим позицию последней из них, а также логическое значение `hasNextPage`, по которому мы определяем, остались ли еще заметки для загрузки. Мы можем рассмотреть схему в GraphQL Playground, чтобы увидеть все доступные варианты данных. Для нашего запроса, скорее всего, потребуется следующая информация:

```
{
  cursor
  hasNextPage
  notes {
    id
    createdAt
    content
    favoriteCount
    author {
      id
      username
      avatar
    }
  }
}
```

Теперь на нашей площадке GraphQL Playground мы можем воплотить их в GraphQL-запрос. Мы напишем его несколько более подробно, чем запросы из главы, посвященной серверу, присвоив ему имя и опциональную переменную `cursor`. Перед использованием GraphQL Playground убедитесь, что сервер API запущен, а затем перейдите по ссылке http://localhost:4000/api. Далее уже в GraphQL Playground добавьте следующий запрос:

```
query noteFeed($cursor: String) {
  noteFeed(cursor: $cursor) {
    cursor
    hasNextPage
    notes {
      id
      createdAt
      content
      favoriteCount
      author {
        username
        id
        avatar
      }
    }
  }
}
```

Там же на площадке добавьте query variable, чтобы протестировать использование переменной:

```
{
  "cursor": ""
}
```

Чтобы проверить эту переменную, замените пустую строку на значение ID любой из заметок в базе данных (рис. 14.1).

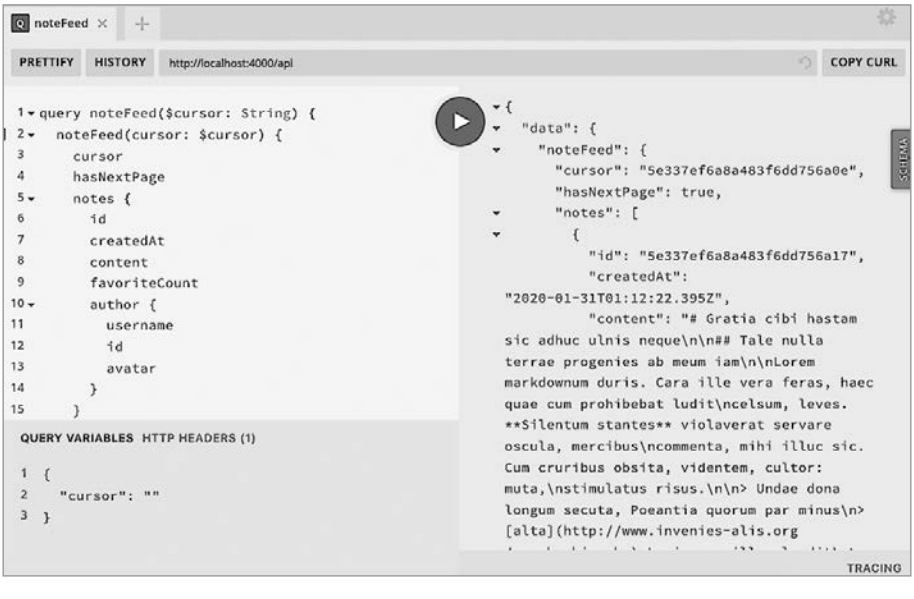

Рис. 14.1. Запрос noteFeed в GraphQL Playground

Теперь, когда мы знаем, что запрос написан правильно, можно уверенно интегрировать его в наше веб-приложение. Импортируйте в файл src/pages/home.js библиотеку `useQuery`, а также GraphQL-синтаксис через библиотеку `qql` из `@apollo/client`.

```
// Импортируем необходимые библиотеки
import { useQuery, gql } from '@apollo/client';

// Наш GraphQL-запрос, хранящийся в виде переменной
const GET_NOTES = gql`
  query NoteFeed($cursor: String) {
    noteFeed(cursor: $cursor) {
      cursor
      hasNextPage
      notes {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;
```

Теперь мы можем интегрировать этот запрос в приложение React. Для этого передадим строку GraphQL-запроса в Apollo-хук `useQuery`. Этот хук вернет объект, содержащий одно из следующих значений:

`data`

Данные, возвращаемые запросом в случае успеха.

`loading`

Состояние загрузки, устанавливаемое как True при получении данных. Это позволяет нам отображать пользователям индикатор загрузки.

`error`

Если получение данных провалилось, в приложение возвращается ошибка.

Мы можем добавить в компонент Home запрос:

```
const Home = () => {
  // Хук запроса
  const { data, loading, error, fetchMore } = useQuery(GET_NOTES);

  // Если данные загружаются, отображаем сообщение о загрузке
```

```
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, отображаем сообщение об ошибке
  if (error) return <p>Error!</p>;

  // Если получение данных прошло успешно, отображаем их в UI
  return (
    <div>
      {console.log(data)}
      The data loaded!
    </div>
  );
};

export default Home;
```

Если все прошло успешно, то на домашней странице приложения вы увидите сообщение The data loaded! (рис. 14.2). Мы также включили инструкцию `console.log`, которая будет выводить данные в консоль браузера. Анализ структуры результатов данных может послужить хорошим ориентиром при их интеграции в приложение.

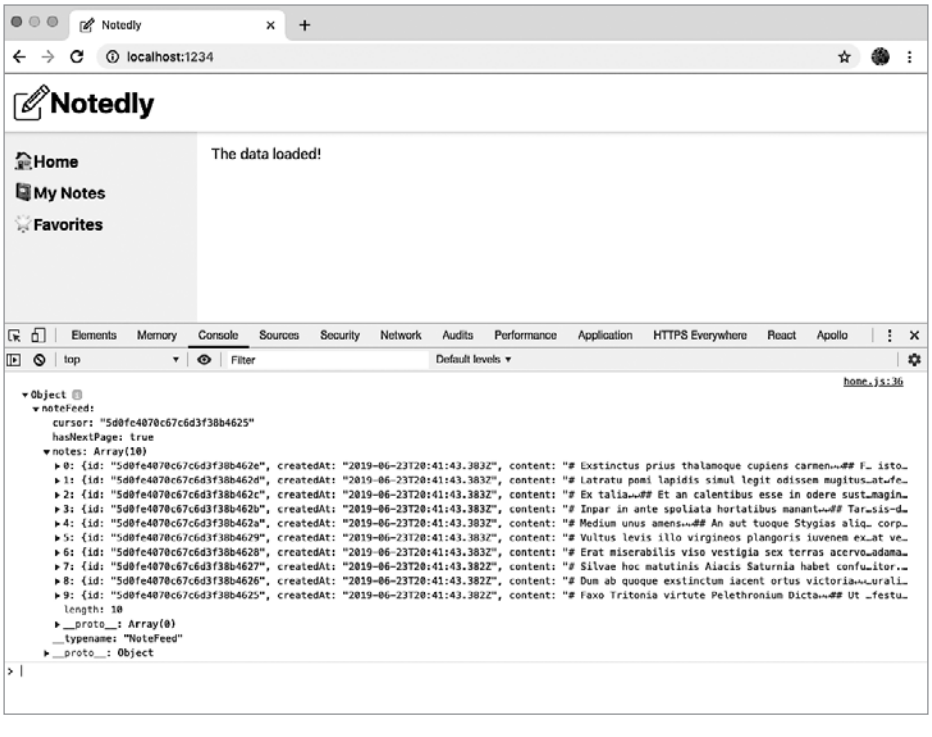

Рис. 14.2. Если данные были получены успешно, компонент отобразит сообщение The data loaded!, а сами данные будут выведены в консоль

Теперь давайте интегрируем полученные данные в приложение. Для этого мы выполним `map` для массива заметок, возвращенных с данными. React требует присвоения каждому результату уникального ключа, в качестве которого мы будем использовать ID отдельных заметок. Для начала мы отобразим имя автора каждой заметки.

```
const Home = () => {
  // Хук запроса
  const { data, loading, error, fetchMore } = useQuery(GET_NOTES);

  // Если данные загружаются, отображаем сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, отображаем сообщение об ошибке
  if (error) return <p>Error!</p>;

  // Если получение данных прошло успешно, отображаем их в UI
  return (
    <div>
      {data.noteFeed.notes.map(note => (
        <div key={note.id}>{note.author.username}</div>
      ))}
    </div>
  );
};
```

ИСПОЛЬЗОВАНИЕ JS-МЕТОДА MAP()

Если вам не доводилось работать с методом `map()`, то его синтаксис сначала может показаться несколько неудобным. Этот метод позволяет выполнять действия для элементов массива. Это может здорово пригодиться при работе с данными, полученными от API, например при отображении каждого элемента в шаблоне определенным образом. Для более подробного ознакомления с `map()` я рекомендую прочесть документацию по MDN (https://oreil.ly/Oca3y).

Если у вас в БД есть данные, то сейчас на странице вы должны видеть список имен пользователей (рис. 14.3).

Выполнив отображение данных, мы можем написать оставшуюся часть компонента. Поскольку наши заметки написаны на Markdown, мы импортируем библиотеку, которая позволит нам отрисовывать этот вид разметки на странице.

В src/pages/home.js:

```
import ReactMarkdown from 'react-markdown';
```

Теперь можно обновить UI, включив аватар автора, его имя, дату создания заметки, число отметок «Избранное» для нее и, наконец, само содержимое.

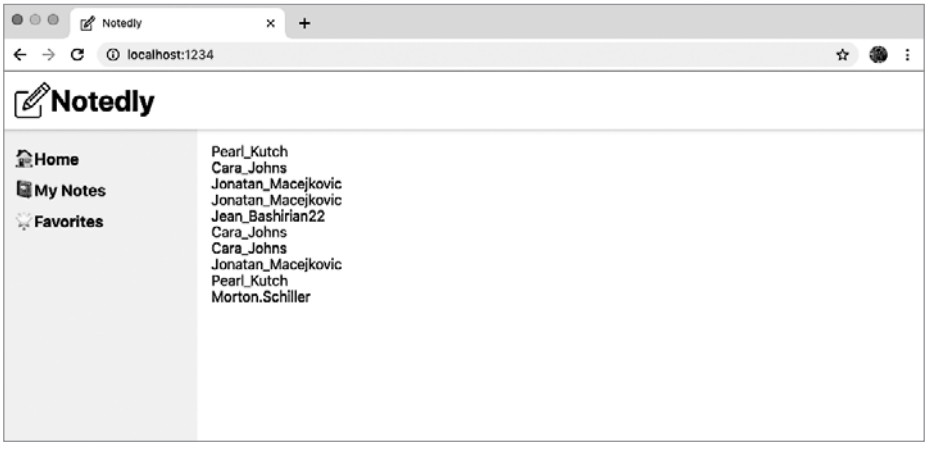

Рис. 14.3. Имена пользователей из нашей базы данных, выведенные на экран

В src/pages/home.js:

```
// Если данные получены успешно, отображаем их в UI
return (
  <div>
    {data.noteFeed.notes.map(note => (
      <article key={note.id}>
      <img
          src={note.author.avatar}
          alt={`${note.author.username} avatar`}
          height="50px"
        />{' '}
        {note.author.username} {note.createdAt} {note.favoriteCount}{' '}
        <ReactMarkdown source={note.content} />
      </article>
    ))}
  </div>
);
```

ПРОБЕЛЫ В REACT

React удаляет пробелы между элементами на новых строках. Использование {' '} в разметке позволит добавлять их вручную.

Теперь вы должны увидеть полный список заметок в браузере. Прежде чем мы перейдем к их стилизации, у нас есть возможность произвести небольшой рефакторинг. Это наша первая страница, отображающая заметки, но, как нам известно, не последняя. На других страницах нам понадобится отображать отдельные заметки, а также ленты других типов (например, my notes или favorites).

Давайте создадим два новых компонента: src/components/Note.js и src/components/NoteFeed.js.

В файл src/components/Note.js мы включим разметку для отдельной заметки. Для этого передадим каждой из функций компонентов свойство с соответствующим содержимым.

```
import React from 'react';
import ReactMarkdown from 'react-markdown';

const Note = ({ note }) => {
  return (
    <article>
      <img
        src={note.author.avatar}
        alt="{note.author.username} avatar"
        height="50px"
      />{' '}
      {note.author.username} {note.createdAt} {note.favoriteCount}{' '}
      <ReactMarkdown source={note.content} />
    </article>
  );
};

export default Note;
```

А теперь для компонента src/components/NoteFeed.js:

```
import React from 'react';
import Note from './Note';

const NoteFeed = ({ notes }) => {
  return (
    <div>
      {notes.map(note => (
        <div key={note.id}>
          <Note note={note} />
        </div>
      ))}
    </div>
  );
};

export default NoteFeed;
```

Наконец, мы можем обновить компонент src/pages/home.js, чтобы он ссылался на наш запрос NoteFeed.

```
import React from 'react';
import { useQuery, gql } from '@apollo/client';

import Button from '../components/Button';
import NoteFeed from '../components/NoteFeed';
```

```
const GET_NOTES = gql`
  query NoteFeed($cursor: String) {
    noteFeed(cursor: $cursor) {
      cursor
      hasNextPage
      notes {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;

const Home = () => {
  // Хук запроса
  const { data, loading, error, fetchMore } = useQuery(GET_NOTES);

  // Если данные загружаются, отображаем сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, отображаем сообщение об ошибке
  if (error) return <p>Error!</p>;

  // Если загрузка данных произошла успешно, отображаем их в UI
   return <NoteFeed notes={data.noteFeed.notes} />;
};

export default Home;
```

После выполнения такого рефакторинга можно легко пересоздавать заметки и экземпляры ленты заметок по всему приложению.

Стиль

Теперь, когда мы прописали компоненты и можем просматривать данные, пришло время добавить стиль. Одним из наиболее очевидных улучшений будет изменение представления даты created at. Для этого мы используем библиотеку `date-fns` (https://date-fns.org), которая предоставляет небольшие компоненты для работы с датами в JS. Импортируйте эту библиотеку в файл src/components.js и примените преобразование, обновив разметки дат.

```
// Импортируем утилиту форматирования из 'date-fns`
import { format } from 'date-fns';

// Обновляем разметку даты, чтобы привести ее в формат Месяц, День, Год
{format(note.createdAt, 'MMM Do YYYY')} Favorites:{' '}
```

Отформатировав дату, мы можем использовать библиотеку Styled Components, чтобы обновить макет заметки.

```
import React from 'react';
import ReactMarkdown from 'react-markdown';
import { format } from 'date-fns';
import styled from 'styled-components';

// Ограничиваем расширение заметок до 800 пикселей
const StyledNote = styled.article`
  max-width: 800px;
  margin: 0 auto;
`;

// Стилизуем метаданные заметки
const MetaData = styled.div`
  @media (min-width: 500px) {
    display: flex;
    align-items: top;
  }
`;

// Добавляем пространство между аватаром и метаданными
const MetaInfo = styled.div`
  padding-right: 1em;
`;

// Выравниваем 'UserActions' по правой стороне на больших экранах
const UserActions = styled.div`
  margin-left: auto;
`;

const Note = ({ note }) => {
  return (
    <StyledNote>
      <MetaData>
        <MetaInfo>
          <img
            src={note.author.avatar}
            alt="{note.author.username} avatar"
            height="50px"
          />
        </MetaInfo>
        <MetaInfo>
          <em>by</em> {note.author.username} <br />
          {format(note.createdAt, 'MMM Do YYYY')}
        </MetaInfo>
        <UserActions>
          <em>Favorites:</em> {note.favoriteCount}
        </UserActions>
      </MetaData>
```

```
      <ReactMarkdown source={note.content} />
    </StyledNote>
  );
};

export default Note;
```

Также следует добавить немного пространства и разделение между заметками в компоненте NoteFeed.js.

```
import React from 'react';
import styled from 'styled-components';

const NoteWrapper = styled.div`
  max-width: 800px;
  margin: 0 auto;
  margin-bottom: 2em;
  padding-bottom: 2em;
  border-bottom: 1px solid #f5f4f0;
`;

import Note from './Note';

const NoteFeed = ({ notes }) => {
  return (
    <div>
      {notes.map(note => (
        <NoteWrapper key={note.id}>
          <Note note={note} />
        </NoteWrapper>
      ))}
    </div>
  );
};

export default NoteFeed;
```

С помощью этих обновлений мы добавили в приложение стили макета.

Динамические запросы

Пока что наше приложение состоит из трех статических маршрутов. Они расположены по статическим URL-адресам и всегда будут выполнять один и тот же запрос данных. Тем не менее приложениям, как правило, требуются динамические маршруты и основанные на них запросы. В качестве примера можно взять твиты в Twitter, каждый из которых получает уникальный URL со ссылкой twitter.com/<username>/status/<tweet_id>. Это дает пользователям возможность обращаться к конкретным твитам, а также делиться ими как с экосистемой Twitter, так и с другими интернет-ресурсами.

В нашем же приложении сейчас обращаться к заметкам можно только через ленту, но мы хотим дать пользователям возможность просматривать и ссылаться на каждую из них по отдельности. Для реализации этого мы настроим динамическую маршрутизацию, а также GraphQL-запрос отдельных заметок. Задача в том, чтобы пользователи могли обращаться к маршрутам по ссылке /note/<note_id>.

Для начала создадим новый компонент страницы в файле src/pages/notes.js. Туда передадим объект `props` (свойство), который через React Router включает свойство `match`. Оно содержит информацию о том, как путь маршрута сопоставляется с URL. Это откроет нам доступ к параметрам URL через `match.params`.

```
import React from 'react';

const NotePage = props => {
  return (
    <div>
      <p>ID: {props.match.params.id}</p>
    </div>
  );
};

export default NotePage;
```

Теперь можно добавить в файл src/pages/index.js соответствующий маршрут. Этот маршрут будет включать параметр ID, определяемый через `:id`.

```
// Импортируем React и зависимости маршрутизации
import React from 'react';
import { BrowserRouter as Router, Route } from 'react-router-dom';

// Импортируем общий компонент макета
import Layout from '../components/Layout';

// Импортируем маршруты
import Home from './home';
import MyNotes from './mynotes';
import Favorites from './favorites';
import NotePage from './note';

// Определяем маршруты
const Pages = () => {
  return (
    <Router>
      <Layout>
        <Route exact path="/" component={Home} />
        <Route path="/mynotes" component={MyNotes} />
        <Route path="/favorites" component={Favorites} />
        <Route path="/note/:id" component={NotePage} />
      </Layout>
    </Router>
```

```
  );
};

export default Pages;
```

Теперь при посещении http://localhost:1234/note/123 на страницу будет выводиться `ID: 123`. Чтобы это проверить, замените параметр ID на значение по своему усмотрению, например /notes/pizza или /note/GONNAPARTYLIKE1999. Это весьма неплохо, но пользы тут мало. Давайте обновим компонент src/pages/note.js, чтобы делать запрос заметки с ID из URL. Для этого мы используем запрос `note` из нашего API, а также React-компонент `Note`.

```
import React from 'react';
// Импортируем зависимости GraphQL
import { useQuery, gql } from '@apollo/client';

// Импортируем компонент Note
import Note from '../components/Note';

// Запрос note, принимающий переменную ID
const GET_NOTE = gql`
  query note($id: ID!) {
    note(id: $id) {
      id
      createdAt
      content
      favoriteCount
      author {
        username
        id
        avatar
      }
    }
  }
`;

const NotePage = props => {
  // Сохраняем id из url в виде переменной
  const id = props.match.params.id;

  // Запрашиваем хук, передавая значение id в качестве переменной
  const { loading, error, data } = useQuery(GET_NOTE, { variables: { id } });

  // Если данные загружаются, отображаем сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, отображаем сообщение об ошибке
  if (error) return <p>Error! Note not found</p>;

  // Если загрузка данных произошла успешно, отображаем их в UI
  return <Note note={data.note} />;
};

export default NotePage;
```

Теперь при переходе по URL с параметром ID будет отображаться либо соответствующая заметка, либо сообщение об ошибке. Наконец, давайте обновим компонент src/comonents/NoteFeed.js, чтобы он отображал ссылку на отдельную заметку в UI.

Сначала из React Router импортируйте в начало файла `{Link}`.

```
import { Link } from 'react-router-dom';
```

Затем включите в JSX ссылку на страницу заметки:

```
<NoteWrapper key={note.id}>
  <Note note={note} />
  <Link to={`note/${note.id}`}>Permalink</Link>
</NoteWrapper>
```

Выполнив это, мы ввели в приложение динамическую маршрутизацию и дали пользователям возможность просматривать отдельные заметки.

Пагинация

Сейчас на домашней странице приложения мы получаем только 10 последних заметок. Если мы хотим отобразить дополнительные заметки, то необходимо применить пагинацию. Как мы уже говорили в начале этой главы и при разработке сервера API, этот API возвращает `cursor`, который является идентификатором последней заметки, выданной на странице результатов. К тому же он возвращает логическое значение `hasNextPage`, которое будет `true`, если в базе данных есть дополнительные заметки для отображения. Делая запрос к API, мы можем передать в качестве аргумента `cursor`, чтобы получить следующие 10 элементов.

Другими словами, если у нас есть список из 25 объектов (с соответствующими ID от 1 до 25), то, когда мы делаем первый запрос, в ответе будут содержаться элементы 1–10, а также `cursor` и `hasNextPage` со значениями 10 и `true` соответственно. Если мы сделаем запрос, передав `cursor` со значением 10, то получим элементы 11–20, а также `cursor` со значением 20, а значение `hasNextPage` будет снова `true`. И наконец, если мы сделаем третий запрос, передав `cursor` со значением 20, то получим элементы 21–25, а также `cursor` со значением 25, но значение `hasNextPage` в этот раз уже будет `false`. Именно эту логику мы будем реализовывать в запросе `noteFeed`.

Для этого давайте обновим файл src/pages/home.js, чтобы делать разбиваемые на страницы запросы. Когда пользователь в UI нажимает Load More, на странице должны загружаться следующие 10 заметок. Нам нужно, чтобы это происходило без обновления самой страницы, для чего нам потребуется включить в компонент запроса аргумент `fetchMore` и отображать компонент `Button`, только когда

hasNextPage будет true. Пока что мы пропишем это напрямую в компонент домашней страницы, но такой код можно будет с легкостью выделить в отдельный компонент или включить в компонент noteFeed.

```
// Если получение данных произошло успешно, отображаем их в UI
return (
  // Добавляем элемент <React.Fragment>, чтобы предоставить родительский
  // элемент
  <React.Fragment>
    <NoteFeed notes={data.noteFeed.notes} />
    {/* Only display the Load More button if hasNextPage is true */}
    {data.noteFeed.hasNextPage && (
      <Button>Load more</Button>
    )}
  </React.Fragment>
);
```

УСЛОВНЫЕ ИНСТРУКЦИИ В REACT

В предыдущем примере мы условно отображаем кнопку Load more, используя встраиваемую инструкцию if с оператором &&. Эта кнопка будет отображаться в случаях, когда hasNextPage будет true. Подробнее об условной отрисовке вы можете прочитать в официальной документации к React (https://oreil.ly/a_F5s).

Теперь мы можем обновить компонент <Button>, чтобы использовать обработчик onClick. Когда пользователь будет кликать по кнопке, мы будем запускать метод fetchMore, совершая дополнительный запрос и дополняя страницу возвращаемыми им данными.

```
{data.noteFeed.hasNextPage && (
  // onClick выполняет запрос, передавая в качестве переменной текущий курсор
  <Button
    onClick={() =>
      fetchMore({
        variables: {
          cursor: data.noteFeed.cursor
        },
        updateQuery: (previousResult, { fetchMoreResult }) => {
          return {
            noteFeed: {
              cursor: fetchMoreResult.noteFeed.cursor,
              hasNextPage: fetchMoreResult.noteFeed.hasNextPage,
              // Совмещаем новые результаты со старыми
              notes: [
                ...previousResult.noteFeed.notes,
                ...fetchMoreResult.noteFeed.notes
              ],
              __typename: 'noteFeed'
            }
```

```
            };
          }
        })
      }
    >
      Load more
    </Button>
  )}
```

Код выше может показаться несколько грубоватым, поэтому давайте разделим его на составляющие и объясним по порядку. Компонент `<Button>` включает обработчик `onClick`. При нажатии на кнопку выполняется новый запрос с помощью метода `fetchMore`, передающего значение `cursor`, полученное из предыдущего запроса. После возврата выполняется `updateQuery`, который обновляет значения `cursor` и `hasNextPage`, а также совмещает результаты в один массив. `__typeName` — это имя запроса, которое включается в результаты Apollo.

С помощью этих изменений мы получили возможность просматривать все заметки из ленты. Можете проверить сами, прокрутив эту ленту до конца. Если ваша БД содержит более 10 заметок, то будет также отображаться кнопка Load More, нажатие на которую будет добавлять на страницу результат следующего запроса `NoteFeed`.

Итоги

В этой главе мы рассмотрели очень многое. Мы настроили работу Apollo Client с нашим приложением React и интегрировали несколько GraphQL-запросов в UI. Мощность GraphQL проявляется в его способности писать отдельные запросы, возвращающие конкретно запрашиваемые через UI данные. В следующей главе мы интегрируем аутентификацию пользователей, позволив им авторизовываться и просматривать заметки, а также раздел «Избранное».

ГЛАВА 15

Аутентификация и состояние

Мы с семьей недавно переехали. После заполнения и подписания множества форм (рука до сих пор болит) нам вручили ключи от парадной двери. Они нужны всякий раз, чтобы открыть дверь и войти, когда мы возвращаемся домой. Я рад, что теперь мне уже не нужно заново заполнять анкету, чтобы попасть к себе в квартиру. При этом я также ценю наличие замка, который охраняет дом от незваных гостей.

Аутентификация на клиентской стороне работает примерно так же. Наши пользователи будут заполнять форму и получать ключи от сайта в форме пароля и токена, хранящегося в их браузере. При возвращении на сайт они либо будут аутентифицированы автоматически посредством токена, либо смогут авторизоваться, используя пароль.

В этой главе мы построим систему веб-аутентификации при помощи нашего GraphQL API. Для этого мы будем создавать формы, сохранять JWT в браузере, отправлять токены с каждым запросом и отслеживать состояние приложения.

Создание формы регистрации

Для начала реализации аутентификации клиента в нашем приложении мы можем создать React-компонент регистрации пользователя. Но перед этим давайте представим принцип работы этого компонента.

Сначала пользователь будет переходить по маршруту приложения `/signup`. На этой странице ему будет представлена форма для ввода имейла, имени и пароля. Отправка такой формы будет выполнять мутацию API `signUp`. Если мутация пройдет успешно, будет создана учетная запись, а API вернет JWT. Если же произойдет ошибка, мы сможем уведомить о ней пользователя. В этом случае мы будем отображать обобщенное сообщение об ошибке, но можно и обновить API, чтобы возвращать конкретные сообщения, например о том, что данное имя пользователя уже занято или что такой имейл уже зарегистрирован.

Начнем мы с организации нового маршрута, но для начала создадим новый компонент React в src/pages/signup.js.

```
import React, { useEffect } from 'react';

// Добавляем props, передаваемый в компонент для дальнейшего использования
const SignUp = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Sign Up — Notedly';
  });

  return (
    <div>
      <p>Sign Up</p>
    </div>
  );
};

export default SignUp;
```

Теперь обновляем список маршрутов в файле src/pages/index.js, включив маршрут `signup`.

```
// Импортируем маршрут signup
import SignUp from './signup';

//Добавляем этот маршрут в компонент Pages
<Route path="/signup" component={SignUp} />
```

Добавив данный маршрут, мы сможем перейти по адресу http://localhost:1234/signup и увидеть практически пустую страницу регистрации. Теперь добавим в нашу форму разметку:

```
import React, { useEffect } from 'react';

const SignUp = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Sign Up — Notedly';
  });

  return (
    <div>
      <form>
        <label htmlFor="username">Username:</label>
        <input
          required
          type="text"
          id="username"
          name="username"
          placeholder="username"
```

```
        />
        <label htmlFor="email">Email:</label>
        <input
          required
          type="email"
          id="email"
          name="email"
          placeholder="Email"
        />
        <label htmlFor="password">Password:</label>
        <input
          required
          type="password"
          id="password"
          name="password"
          placeholder="Password"
        />
        <button type="submit">Submit</button>
      </form>
    </div>
  );
};

export default SignUp;
```

HTMLFOR

Если вы только начали изучать React, то один из распространенных подводных камней заключается в JSX-атрибутах, которые отличаются от своих аналогов в HTML. В данном случае мы используем JSX-атрибут `htmlFor` вместо его HTML-собрата `for`, чтобы избежать коллизий с JavaScript. Короткий, но полный список этих атрибутов можно найти в разделе DOM Elements документации React (https://oreil.ly/Kn5Ke).

Теперь можно добавить стили, импортировав компонент `Button` и определив форму как стилизованный компонент:

```
import React, { useEffect } from 'react';
import styled from 'styled-components';

import Button from '../components/Button';

const Wrapper = styled.div`
  border: 1px solid #f5f4f0;
  max-width: 500px;
  padding: 1em;
  margin: 0 auto;
`;

const Form = styled.form`
  label,
```

```
  input {
    display: block;
    line-height: 2em;
  }

  input {
    width: 100%;
    margin-bottom: 1em;
  }
`;

const SignUp = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Sign Up — Notedly';
  });

  return (
    <Wrapper>
      <h2>Sign Up</h2>
      <Form>
        <label htmlFor="username">Username:</label>
        <input
          required
          type="text"
          id="username"
          name="username"
          placeholder="username"
        />
        <label htmlFor="email">Email:</label>
        <input
          required
          type="email"
          id="email"
          name="email"
          placeholder="Email"
        />
        <label htmlFor="password">Password:</label>
        <input
          required
          type="password"
          id="password"
          name="password"
          placeholder="Password"
        />
        <Button type="submit">Submit</Button>
      </Form>
    </Wrapper>
  );
};

export default SignUp;
```

Формы и состояния в React

В приложении происходят изменения. В форму вводятся данные, пользователь переключает ползунок, сообщение отправляется. В React мы можем отслеживать эти изменения на уровне компонентов, назначая состояние. В форме нам нужно отслеживать состояние каждого элемента, чтобы можно было его отправлять.

ХУКИ REACT

В этой книге мы используем функциональные компоненты и обновленный API Hooks. Если вы работали с другими обучающими ресурсами, использующими React-компоненты `class`, то здесь все может выглядеть для вас несколько иначе. Больше информации о хуках вы можете получить в документации React (https://oreil.ly/Tz9Hg).

Чтобы начать работу с состояниями, сначала обновим импорт React в файле src/pages/signup.js, добавив `useState`.

```
import React, { useEffect, useState } from 'react';
```

Затем определим предустановленное значение формы в компоненте `SignUp`.

```
const SignUp = props => {
  // Устанавливаем состояние формы по умолчанию
  const [values, setValues] = useState();

// Остальная часть компонента
};
```

Теперь обновим компонент так, чтобы он изменял состояние формы при заполнении ее полей и выполнял операцию при ее отправке пользователем. Сначала создадим функцию `onChange`, которая будет обновлять состояние компонента при каждом обновлении формы. Мы также обновим разметку каждого элемента формы, чтобы вызывать эту функцию при каждом внесении изменений пользователем с помощью свойства `onChange`. Затем мы включим в элемент `form` обработчик `onSubmit`. Пока что мы просто запишем эти формы в консоль.

В /src/pages/sigunp.js:

```
const SignUp = () => {
  // Устанавливаем состояние формы по умолчанию
  const [values, setValues] = useState();

  // Обновляем состояние при вводе пользователем данных
  const onChange = event => {
    setValues({
      ...values,
      [event.target.name]: event.target.value
    });
  };
```

```
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Sign Up — Notedly';
  });

  return (
    <Wrapper>
      <h2>Sign Up</h2>
      <Form
        onSubmit={event => {
          event.preventDefault();
          console.log(values);
        }}
        >
        <label htmlFor="username">Username:</label>
        <input
          required
          type="text"
          name="username"
          placeholder="username"
          onChange={onChange}
        />
        <label htmlFor="email">Email:</label>
        <input
          required
          type="email"
          name="email"
          placeholder="Email"
          onChange={onChange}
        />
        <label htmlFor="password">Password:</label>
        <input
          required
          type="password"
          name="password"
          placeholder="Password"
          onChange={onChange}
        />
        <Button type="submit">Submit</Button>
      </Form>
    </Wrapper>
  );
};
```

Добавив разметку, мы можем запрашивать данные с помощью GraphQL-мутации.

Мутация signUp

Для регистрации пользователя мы будем использовать API-мутацию signUp. Она будет принимать имейл, имя пользователя и пароль в виде переменных, возвращая в случае успеха JWT. Давайте напишем эту мутацию и интегрируем ее в форму регистрации.

Сначала нам нужно импортировать библиотеки Apollo. Будем использовать хуки `useMutation` и `useApolloClient`, а также синтаксис `qql` из Apollo Client. В файле src/pages/signUp рядом с остальными инструкциями импорта библиотек добавьте следующее:

```
import { useMutation, useApolloClient, gql } from '@apollo/client';
```

Теперь напишите GraphQL-мутацию:

```
const SIGNUP_USER = gql`
  mutation signUp($email: String!, $username: String!, $password: String!) {
     signUp(email: $email, username: $username, password: $password)
  }
`;
```

Написав мутацию, мы можем обновить разметку компонента React так, чтобы выполнять эту мутацию при отправке пользователем формы, передавая ее элементы в качестве переменных. Пока что мы будем выводить ответ (которым в случае успеха будет JWT) в консоль.

```
const SignUp = props => {
  // useState, onChange и useEffect остаются здесь без изменений

  //Добавляем хук мутации
  const [signUp, { loading, error }] = useMutation(SIGNUP_USER, {
    onCompleted: data => {
      // Когда мутация завершена, выводим в консоль JSON Web Token
      console.log(data.signUp);
    }
  });

  // Отрисовываем форму
  return (
    <Wrapper>
      <h2>Sign Up</h2>
      {/* Когда пользователь отправляет форму, передаем ее данные в мутацию */}
      <Form
        onSubmit={event => {
          event.preventDefault();
          signUp({
            variables: {
              ...values
            }
          });
        }}
      >
        {/* ... остаток формы остается без изменений ... */}
      </Form>
    </Wrapper>
  );
};
```

Если теперь вы заполните и отправите форму, то должны увидеть в консоли JWT (рис. 15.1). Кроме того, если на площадке GraphQL выполнить запрос `users`, то можно увидеть новую учетную запись (рис. 15.2).

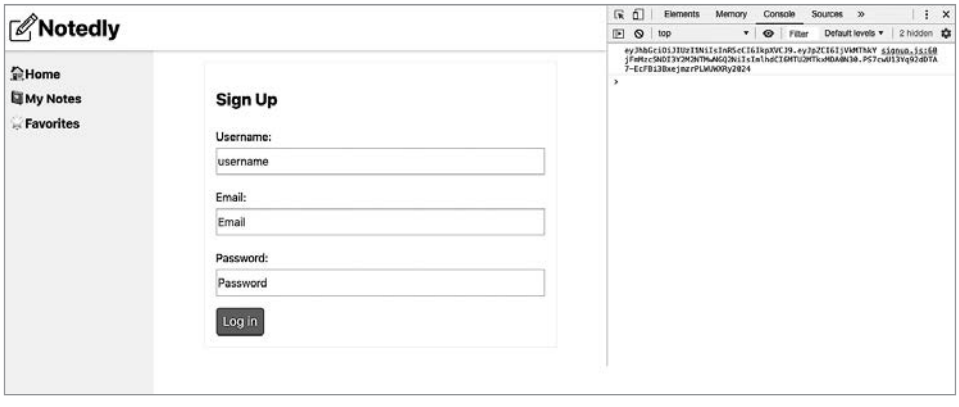

Рис. 15.1. В случае успеха при отправке формы в консоль будет выведен JSON Web Token

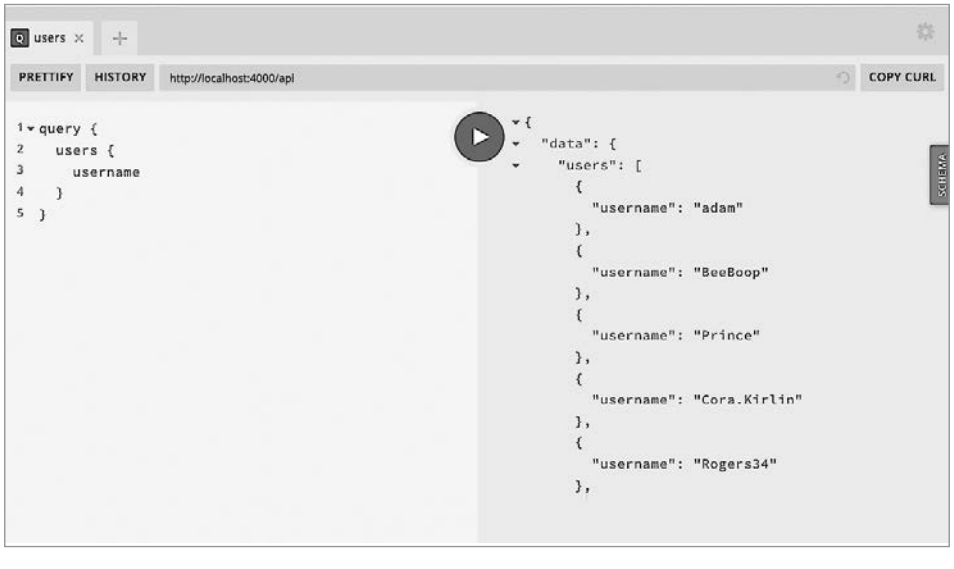

Рис. 15.2. Можно также увидеть список пользователей, выполнив в GraphQL Playground запрос users

Создав мутацию и наладив получение ожидаемых данных, мы займемся сохранением получаемого ответа.

JSON Web Token и локальное хранилище

В случае успеха мутация `signUp` возвращает JWT. Вы можете вспомнить раздел API, где говорилось о том, что JWT (https://jwt.io) позволяет безопасно хранить ID пользователя на его устройстве. Чтобы реализовать эту возможность в клиентском браузере, будем хранить токен в его `localStorage` — простом хранилище пар «ключ–значение», которое сохраняет содержимое на протяжении ряда сессий браузера, пока не будет обновлено или очищено. Давайте изменим нашу мутацию, чтобы она сохраняла токен в `localStorage`.

Обновите хук `useMutation` в src/pages/signup.js, чтобы он сохранял токен в `localStorage` (рис. 15.3).

```
const [signUp, { loading, error }] = useMutation(SIGNUP_USER, {
  onCompleted: data => {
    // Сохраняем JWT в localStorage
    localStorage.setItem('token', data.signUp);
  }
});
```

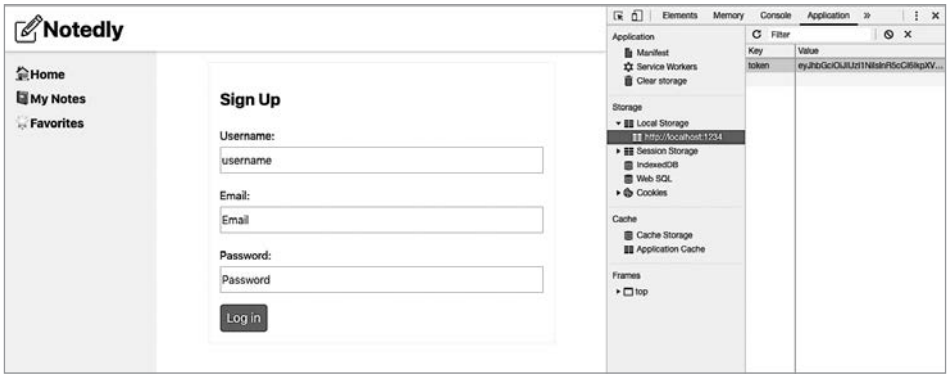

Рис. 15.3. Теперь наш веб-токен сохраняется в localStorage браузера

JWT И БЕЗОПАСНОСТЬ

Когда токен хранится в `localStorage`, любой JS-код, который может выполняться на этой странице, имеет к нему доступ, что делает его уязвимым для атак межсайтового скриптинга (XSS). По этой причине при хранении данных токена в `localStorage` вам нужно уделить внимание ограничению (или избежанию) сценариев, размещенных в CDN. Если сторонний сценарий будет скомпрометирован, он получит доступ к вашему JWT.

Организовав локальное хранение JWT, мы подготовились к его использованию в GraphQL-мутациях и запросах.

Переадресация

Пока что, когда пользователь завершает заполнение регистрационной формы, она отрисовывается повторно в пустом виде. Это не дает пользователю никаких визуальных сигналов о том, что регистрация прошла успешно. Давайте вместо этого будем перенаправлять наших пользователей на домашнюю страницу приложения. Другой вариант — создать страницу Success, которая будет благодарить человека за регистрацию и перенаправлять его в приложение.

Как отмечалось в предыдущих разделах главы, мы передаем свойства в компонент. Мы можем переадресовать маршрут, используя инструкцию `history` маршрутизатора React, которая будет доступна через `props.history.push`. Чтобы это реализовать, мы обновим в мутации событие `onCompleted`, добавив в него переадресацию.

```
const [signUp, { loading, error }] = useMutation(SIGNUP_USER, {
    onCompleted: data => {
      // Сохраняем токен
      localStorage.setItem('token', data.signUp);
      // Перенаправляем пользователя на домашнюю страницу
      props.history.push('/');
    }
  });
```

Теперь пользователи после регистрации будут перенаправляться на домашнюю страницу приложения.

Прикрепление заголовков к запросам

Несмотря на то что мы храним токен в `localStorage`, наш API пока не имеет к нему доступа. Это означает, что даже если пользователь создаст учетную запись, у API не будет возможности идентифицировать этого пользователя. Как пояснялось в разделе по разработке API, каждый вызов API получает токен в заголовке запроса. Поэтому мы изменим клиент так, чтобы он отправлял JWT в качестве заголовка с каждым запросом.

Мы обновим зависимости в файле src/App.js, добавив `createHttpLink` из Apollo Client, а также `setContext` из Apollo-пакета Link Context. После этого мы изменим конфигурацию Apollo, чтобы отправлять токен в заголовке каждого запроса.

```
// Импортируем зависимости Apollo
import {
  ApolloClient,
  ApolloProvider,
  createHttpLink,
  InMemoryCache
} from '@apollo/client';
```

```
import { setContext } from 'apollo-link-context';

// Настраиваем API URI и кэш
const uri = process.env.API_URI;
const httpLink = createHttpLink({ uri });
const cache = new InMemoryCache();

// Проверяем наличие токена и возвращаем заголовки в контекст
const authLink = setContext((_, { headers }) => {
  return {
    headers: {
      ...headers,
      authorization: localStorage.getItem('token') || ''
    }
  };
});

// Создаем клиент Apollo
const client = new ApolloClient({
  link: authLink.concat(httpLink),
  cache,
  resolvers: {},
  connectToDevTools: true
});
```

Теперь мы сможем передавать информацию об авторизованном пользователе в наш API.

Управление локальным состоянием

Мы рассмотрели управление состоянием внутри компонента, но что насчет самого приложения? Бывают случаи, когда информация, используемая компонентами совместно, весьма кстати. Можно передать `props` из базового компонента по всему приложению, но по мере прохождения очередных уровней подкомпонентов это будет приводить к все большему беспорядку. Такие библиотеки, как Redux (https://redux.js.org) и MobX (https://mobx.js.org), служат для решения подобных задач управления состоянием и уже доказали свою эффективность. В нашем случае мы уже используем библиотеку Apollo Client, включающую возможность применять GraphQL-запросы для управления локальным состоянием. Поэтому вместо ввода очередной зависимости мы с вами реализуем свойство локального состояния, которое будет хранить информацию о подтверждении авторизации пользователя.

Библиотека Apollo Client помещает в контекст экземпляр `ApolloClient`, но иногда нам может понадобиться обратиться к нему напрямую. Это можно сделать с помощью хука `useApolloClient`, который позволит выполнять такие действия, как прямое обновление/сброс кэша или запись локальных данных.

Сейчас у нас есть два способа для определения авторизации пользователя в приложении. Во-первых, мы узнаем об этом, когда пользователь успешно отправил регистрационную форму. Во-вторых, мы знаем, что если посетитель обращается к сайту с токеном, хранящимся в `localStorage`, значит, он уже авторизован. Давайте начнем с добавления состояния, когда пользователь завершает заполнение регистрационной формы. Для реализации этого мы будем производить непосредственную запись в локальное хранилище ApolloClient, используя `client.writeData` и хук `useApolloClient`.

Сначала нужно включить хук `useApolloClient` в импорт библиотеки `@apollo/client` в файле src/pages/signup.js.

```
import { useMutation, useApolloClient } from '@apollo/client';
```

В src/pages/signup.js мы будем вызывать функцию `useApolloClient` и обновлять мутацию, в финале добавляя ее данные в локальное хранилище при помощи `writeData`.

```
// Apollo Client
const client = useApolloClient();
// Хук мутации
const [signUp, { loading, error }] = useMutation(SIGNUP_USER, {
  onCompleted: data => {
    // Сохраняем токен
    localStorage.setItem('token', data.signUp);
    // Обновляем локальный кэш
    client.writeData({ data: { isLoggedIn: true } });
    // Перенаправляем пользователя на домашнюю страницу
    props.history.push('/');
  }
});
```

Теперь давайте обновим приложение, чтобы проверять наличие ранее созданного токена при загрузке страницы и обновлять состояние в случае его обнаружения. Сначала обновите конфигурацию `ApolloClient` в src/App.js в пустой объект `resolvers`. Это позволит нам выполнять GraphQL-запросы к локальному кэшу.

```
// Создаем клиент Apollo
const client = new ApolloClient({
  link: authLink.concat(httpLink),
  cache,
  resolvers: {},
  connectToDevTools: true
});
```

Далее мы можем выполнить проверку при начальной загрузке страницы приложения.

```
// Проверяем наличие локального токена
const data = {
  isLoggedIn: !!localStorage.getItem('token')
};

// Записываем данные кэша при начальной загрузке
cache.writeData({ data });
```

А вот здесь самое интересное: теперь мы можем обращаться к `isLoggedin` в виде GraphQL-запроса в любом месте приложения, используя директиву `@client`. Давайте для наглядности обновим заголовок приложения, чтобы он отображал ссылки «Sign Up» и «Sign In», когда `isLoggedIn` будет `false`, и Log Out, если `isLoggedIn` будет `true`.

Импортируйте в файл src/components/Header.js необходимые зависимости и напишите запрос:

```
// Новые зависимости
import { useQuery, gql } from '@apollo/client';
import { Link } from 'react-router-dom';

// Локальный запрос
const IS_LOGGED_IN = gql`
  {
    isLoggedIn @client
  }
`;
```

Теперь мы можем добавить в компонент React простой запрос, чтобы получать состояние вместе с тернарным оператором, отображающим варианты либо для выхода из системы (Log Out), либо для входа (Sign In).

```
const UserState = styled.div`
  margin-left: auto;
`;

const Header = props => {
  // Хук запроса для проверки состояния авторизации пользователя
  const { data } = useQuery(IS_LOGGED_IN);

  return (
    <HeaderBar>
      <img src={logo} alt="Notedly Logo" height="40" />
      <LogoText>Notedly</LogoText>
      {/* Если авторизован, отображаем ссылку logout, в противном
       случае отображаем варианты sign in и sign up */}
      <UserState>
        {data.isLoggedIn ? (
          <p>Log Out</p>
        ) : (
          <p>
```

```
            <Link to={'/signin'}>Sign In</Link> or{' '}
            <Link to={'/signup'}>Sign Up</Link>
          </p>
        )}
      </UserState>
    </HeaderBar>
  );
};
```

Теперь при авторизации пользователь будет видеть вариант Log Out; в противном случае ему будет предложен вариант регистрации или авторизации, и все это благодаря локальному состоянию. При этом мы не ограничены простой булевой логикой. С помощью Apollo мы можем писать локальные распознаватели и определения типов, задействуя все доступные преимущества GraphQL в отношении локального состояния.

Выход из системы

Пока что, выполнив авторизацию, пользователь не может выйти из приложения. Давайте преобразуем выражение Log Out из заголовка в кнопку, которая при нажатии будет отключать пользователя от системы. Чтобы это осуществить, мы при нажатии этой кнопки будем удалять токен из `localStorage`. Для этого мы используем удобный элемент `<button>`, поскольку он не только служит схематическим представлением действия пользователя, но еще и получает направление фокуса в виде, например, ссылки, когда пользователь перемещается по приложению с помощью клавиатуры.

Прежде чем заняться кодом, давайте напишем компонент, который будет отображать кнопку в виде ссылки. Создайте файл src/components/ButtonAsLink.js и добавьте в него следующее:

```
import styled from 'styled-components';

const ButtonAsLink = styled.button`
  background: none;
  color: #0077cc;
  border: none;
  padding: 0;
  font: inherit;
  text-decoration: underline;
  cursor: pointer;

  :hover,
  :active {
    color: #004499;
  }
`;

export default ButtonAsLink;
```

Теперь мы можем реализовать функциональность выхода из приложения в файле src/components/Header.js. Для обработки переадресации нам понадобится использовать компонент высшего порядка `withRouter` маршрутизатора React, так как файл Header.js является компонентом UI, а не определенным маршрутом. Давайте начнем с импорта компонентов `ButtonAsLink` и `withRouter`.

```
// Импортируем Link и withRouter из React Router
import { Link, withRouter } from 'react-router-dom';
// Импортируем компонент ButtonAsLink
import ButtonAsLink from './ButtonAsLink';
```

После этого мы обновим компонент в JSX, включив в него параметр `props`, а также изменим разметку выхода из системы, сделав ее кнопкой.

```
const Header = props => {
  // хук запроса проверки состояния авторизации пользователя,
  // включая клиент для обращения к хранилищу Apollo
  const { data, client } = useQuery(IS_LOGGED_IN);

  return (
    <HeaderBar>
      <img src={logo} alt="Notedly Logo" height="40" />
      <LogoText>Notedly</LogoText>
      {/* Если авторизован, отображаем ссылку log out, в противном
       случае отображаем варианты sign in и sign up */}
      <UserState>
        {data.isLoggedIn ? (
          <ButtonAsLink>
            Logout
          </ButtonAsLink>
        ) : (
          <p>
            <Link to={'/signin'}>Sign In</Link> or{' '}
            <Link to={'/signup'}>Sign Up</Link>
          </p>
        )}
      </UserState>
    </HeaderBar>
  );
};

// Обертываем компонент в компонент высшего порядка withRouter
export default withRouter(Header);
```

WITHROUTER

Когда нам нужно добавить маршрутизацию в компонент, который сам по себе не может выступать в роли маршрута, необходимо использовать компонент высшего порядка `withRouter` маршрутизатора React.

При выходе пользователя из приложения нам нужно сбросить хранилище кэша, чтобы предотвратить нежелательное появление данных вне сессии. Apollo предлагает возможность вызова функции `resetStore`, которая полностью очистит кэш. Давайте добавим в кнопку нашего компонента обработчик `onClick`, чтобы удалить токен пользователя, сбросить хранилище Apollo, обновить локальное состояние и перенаправить пользователя на домашнюю страницу. Для этого обновим хук `useQuery`, добавив в него ссылку на клиент и обернув компонент в инструкцию `export` компонента `withRouter`.

```
const Header = props => {
  // Хук запроса состояния авторизации пользователя
  const { data, client } = useQuery(IS_LOGGED_IN);

  return (
    <HeaderBar>
      <img src={logo} alt="Notedly Logo" height="40" />
      <LogoText>Notedly</LogoText>
      {/* Если авторизован, отображаем ссылку log out, в противном
       случае отображаем варианты sign in и sign up */}
      <UserState>
        {data.isLoggedIn ? (
          <ButtonAsLink
            onClick={() => {
              // Удаляем токен
              localStorage.removeItem('token');
              // Очищаем кэш приложения
              client.resetStore();
              // Обновляем локальное состояние
              client.writeData({ data: { isLoggedIn: false } });
              // Перенаправляем пользователя на домашнюю страницу
              props.history.push('/');
            }}
          >
            Logout
          </ButtonAsLink>
        ) : (
          <p>
            <Link to={'/signin'}>Sign In</Link> or{' '}
            <Link to={'/signup'}>Sign Up</Link>
          </p>
        )}
      </UserState>
    </HeaderBar>
  );
};

export default withRouter(Header);
```

Наконец, нам нужно, чтобы при сбросе хранилища Apollo добавил состояние пользователя обратно в кэшированное состояние. Обновите настройки кэша в файле src/App.js, включив `onResetStore`.

```
// Проверяем наличие локального токена
const data = {
  isLoggedIn: !!localStorage.getItem('token')
};

// Записываем кэшированные данные при начальной загрузке
cache.writeData({ data });
// Записываем данные кэша после его сброса
client.onResetStore(() => cache.writeData({ data }));
```

Теперь авторизованные пользователи смогут легко выходить из приложения. Мы интегрировали эту функциональность напрямую в компонент `Header`, но в будущем можем выделить ее в самостоятельный компонент.

Создание формы авторизации

Сейчас пользователи могут регистрироваться в приложении и выходить из него, но не имеют возможности снова в него войти. Давайте создадим форму авторизации и параллельно произведем небольшой рефакторинг, чтобы повторно использовать большую часть кода, находящуюся в компоненте авторизации.

Первым шагом будет создание нового компонента страницы, который мы поместим в src/pages/signin.js. В этот новый файл добавьте следующее:

```
import React, { useEffect } from 'react';

  const SignIn = props => {
    useEffect(() => {
      // Обновляем заголовок документа
      document.title = 'Sign In — Notedly';
    });

    return (
      <div>
        <p>Sign up page</p>
      </div>
    );
  };

  export default SignIn;
```

Теперь мы можем использовать эту страницу в качестве маршрута, чтобы пользователи могли на нее переходить. Импортируйте страницу маршрута в файл src/pages/index.js и добавьте новый путь маршрута.

```
// Импортируем компонент страницы sign-in
import SignIn from './signin';

const Pages = () => {
```

```
  return (
    <Router>
      <Layout>
        // ... остальные маршруты
        // Добавляем в их список маршрут signin
        <Route path="/signin" component={SignIn} />
      </Layout>
    </Router>
  );
};
```

Прежде чем приступить к реализации формы авторизации, предлагаю прерваться и рассмотреть варианты. Мы можем повторно реализовать форму, во многом аналогично той, что писали для страницы Sign Up, но это окажется трудоемким процессом и в дальнейшем потребует от нас обслуживания двух похожих форм. При изменении одной формы нам понадобится обязательно изменить и другую. Второй вариант — выделить новую форму в собственный компонент, который позволит нам повторно использовать общий код и внести изменения только в одно место. Именно так я и предлагаю сделать.

Сначала мы создадим новый компонент src/components/UserForm.js, в котором расположим разметку `<form>` и стили. Мы внесем в эту форму несколько небольших, но важных изменений, чтобы использовать свойства, которые она получает от родительского компонента. Во-первых, мы переименуем мутацию `onSubmit` в `props.action`, что позволит нам передавать ее в нашу форму через свойства компонента. Во-вторых, мы добавим несколько условных инструкций в тех местах, где наши формы будут отличаться. Мы задействуем второе свойство, назвав его `formType`, которое будет передавать строку. Мы можем изменить отрисовку нашего шаблона на основе значения этой строки.

Напишем мы все это как встроенную инструкцию `if` с логическим оператором `&&` либо как условный тернарный оператор.

```
import React, { useState } from 'react';
import styled from 'styled-components';

import Button from './Button';

const Wrapper = styled.div`
  border: 1px solid #f5f4f0;
  max-width: 500px;
  padding: 1em;
  margin: 0 auto;
`;

const Form = styled.form`
  label,
  input {
    display: block;
```

```
    line-height: 2em;
  }

  input {
    width: 100%;
    margin-bottom: 1em;
  }
`;

  const UserForm = props => {
    // Устанавливаем состояние формы по умолчанию
    const [values, setValues] = useState();

    // Обновляем состояние, когда пользователь вводит данные в форму
    const onChange = event => {
      setValues({
        ...values,
        [event.target.name]: event.target.value
      });
    };

    return (
      <Wrapper>
        {/* Отображаем соответствующий заголовок формы */}
        {props.formType === 'signup' ? <h2>Sign Up</h2> : <h2>Sign In</h2>}
        {/* Выполняем мутацию, когда пользователь отправляет форму */}
        <Form
          onSubmit={e => {
            e.preventDefault();
            props.action({
              variables: {
                ...values
              }
            });
          }}
        >
          {props.formType === 'signup' && (
            <React.Fragment>
              <label htmlFor="username">Username:</label>
              <input
                required
                type="text"
                id="username"
                name="username"
                placeholder="username"
                onChange={onChange}
              />
            </React.Fragment>
          )}
          <label htmlFor="email">Email:</label>
          <input
```

```
          required
          type="email"
          id="email"
          name="email"
          placeholder="Email"
          onChange={onChange}
        />
        <label htmlFor="password">Password:</label>
        <input
          required
          type="password"
          id="password"
          name="password"
          placeholder="Password"
          onChange={onChange}
        />
        <Button type="submit">Submit</Button>
      </Form>
    </Wrapper>
  );
};

export default UserForm;
```

Теперь можно упростить компонент src/pages/signup.js, чтобы использовать компонент общей формы.

```
import React, { useEffect } from 'react';
import { useMutation, useApolloClient, gql } from '@apollo/client';

import UserForm from '../components/UserForm';

const SIGNUP_USER = gql`
  mutation signUp($email: String!, $username: String!, $password: String!) {
     signUp(email: $email, username: $username, password: $password)
  }
`;

const SignUp = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Sign Up — Notedly';
  });

  const client = useApolloClient();
  const [signUp, { loading, error }] = useMutation(SIGNUP_USER, {
    onCompleted: data => {
      // Сохраняем токен
      localStorage.setItem('token', data.signUp);
     // Обновляем локальный кэш
      client.writeData({ data: { isLoggedIn: true } });
      // Перенаправляем пользователя на домашнюю страницу
```

```
      props.history.push('/');
    }
  });

  return (
    <React.Fragment>
      <UserForm action={signUp} formType="signup" />
      {/* Если данные загружаются, отображаем сообщение о загрузке */}
      {loading && <p>Loading...</p>}
      {/* Если при загрузке произошел сбой, отображаем сообщение об ошибке */}
      {error && <p>Error creating an account!</p>}
    </React.Fragment>
  );
};

export default SignUp;
```

Наконец, можно написать компонент `SignIn`, используя мутацию `signIn` и компонент `UserForm`. В src/pages/signin.js:

```
import React, { useEffect } from 'react';
import { useMutation, useApolloClient, gql } from '@apollo/client';

import UserForm from '../components/UserForm';

const SIGNIN_USER = gql`
  mutation signIn($email: String, $password: String!) {
    signIn(email: $email, password: $password)
  }
`;

const SignIn = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Sign In — Notedly';
  });

  const client = useApolloClient();
  const [signIn, { loading, error }] = useMutation(SIGNIN_USER, {
    onCompleted: data => {
      // Сохраняем токен
      localStorage.setItem('token', data.signIn);
      // Обновляем локальный кэш
      client.writeData({ data: { isLoggedIn: true } });
      // Перенаправляем пользователя на домашнюю страницу
      props.history.push('/');
    }
  });

  return (
    <React.Fragment>
      <UserForm action={signIn} formType="signIn" />
```

```
      {/* Если данные загружаются, отображаем сообщение о загрузке */}
      {loading && <p>Loading...</p>}
      {/* Если при загрузке произошел сбой, отображаем сообщение об ошибке */}
      {error && <p>Error signing in!</p>}
    </React.Fragment>
  );
};

export default SignIn;
```

Теперь у нас есть обслуживаемый компонент формы, а наши пользователи смогут не только регистрироваться, но и авторизовываться в приложении.

Защищенные маршруты

Стандартный шаблон приложения подразумевает ограничение доступа авторизованных пользователей к определенным страницам или разделам сайта. В нашем случае неавторизованные пользователи не смогут пользоваться страницами My Notes или Favorites. Мы можем реализовать этот шаблон в маршрутизаторе, автоматически направляя неавторизованных пользователей на страницу Sign In, если они будут пробовать перейти по этим маршрутам.

В файле src/pages/index.js начнем с импорта необходимых зависимостей и добавления запроса `isLoggedIn`.

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

const IS_LOGGED_IN = gql`
  {
    isLoggedIn @client
  }
`;
```

Далее импортируем библиотеку `Redirect` маршрутизатора React и напишем компонент `PrivateRoute`, который будет перенаправлять пользователей, если они не авторизованы.

```
// Добавляем Redirect в импорт react-router
import { BrowserRouter as Router, Route, Redirect } from 'react-router-dom';

// Добавляем компонент PrivateRoute под компонентом 'Pages'
const PrivateRoute = ({ component: Component, ...rest }) => {
  const { loading, error, data } = useQuery(IS_LOGGED_IN);
  // Если данные загружаются, выводим сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, выводим сообщение об ошибке
  if (error) return <p>Error!</p>;
  // Если пользователь авторизован, направляем его к запрашиваемому компоненту
```

```
  // В противном случае перенаправляем на страницу авторизации
  return(
    <Route
      {...rest}
      render={props =>
        data.isLoggedIn === true ? (
          <Component {...props} />
        ) : (
          <Redirect
            to={{
              pathname: '/signin',
              state: { from: props.location }
            }}
          />
        )
      }
    />
  );
};

export default Pages;
```

Наконец, мы можем обновить любые из маршрутов, предназначенных для авторизованных пользователей, чтобы они использовали компонент `PrivateRoute`.

```
const Pages = () => {
  return (
    <Router>
      <Layout>
        <Route exact path="/" component={Home} />
        <PrivateRoute path="/mynotes" component={MyNotes} />
        <PrivateRoute path="/favorites" component={Favorites} />
        <Route path="/note/:id" component={Note} />
        <Route path="/signup" component={SignUp} />
        <Route path="/signin" component={SignIn} />
      </Layout>
    </Router>
  );
};
```

СОСТОЯНИЕ ПЕРЕАДРЕСАЦИИ

Когда мы переадресуем приватный маршрут, мы также сохраняем соответствующий ему URL как состояние. Это позволяет перенаправлять пользователей обратно на страницу, на которую они пытались перейти изначально. Мы можем обновить переадресацию на страницу авторизации, чтобы при необходимости активировать эту функцию с помощью `props.state.location.from`.

Теперь, когда пользователь попытается перейти на страницу, предназначенную для авторизованных пользователей, он будет перенаправлен на страницу Sign In.

Итоги

В этой главе мы рассмотрели два важнейших принципа построения клиентских JS-приложений: аутентификацию и состояние. Я надеюсь, что, создавая этот полноценный поток аутентификации, вы разобрались, как работают учетные записи пользователей с клиентским приложением. С этого момента я рекомендую уделить время изучению альтернативных вариантов вроде OAuth и таких сервисов аутентификации, как Auth0, Okta и Firebase. Кроме того, в пройденной главе вы научились управлять состоянием как на уровне компонентов при помощи React Hooks API, так и в приложении, используя локальное состояние Apollo. С помощью этих ключевых принципов вы можете строить надежные приложения UI.

ГЛАВА 16

Операции создания, чтения, изменения и удаления

Мне нравятся бумажные блокноты, я постоянно ношу такие при себе. Стоят они обычно недорого, и я быстро заполняю их набросками идей. Не так давно я купил блокнот подороже в твердом переплете с красивой обложкой и разноцветной бумагой. В момент покупки у меня были грандиозные планы на этот блокнот, но в итоге он так и пролежал несколько месяцев абсолютно пустой на столе. В конце концов я убрал его на полку и вернулся к привычному варианту блокнота.

Как и мой «модный» блокнот, наше приложение полезно, только если с ним можно взаимодействовать. Вы можете вспомнить раздел по разработке API, где говорилось, что Notedly — это приложение типа CRUD («создание, чтение, изменение и удаление»). Авторизованный пользователь сможет создавать и читать заметки, редактировать их содержимое, изменять статус на «Избранное», ну и, конечно же, удалять. В этой главе мы реализуем всю эту функциональность в рамках пользовательского веб-интерфейса, используя GraphQL-мутации и запросы.

Создание заметок

Пока что у нас есть средства для просмотра заметок, но не для создания. Это как иметь блокнот без ручки. Поэтому давайте добавим возможность создания заметок пользователями. Для этого мы создадим форму `textArea`, в которой пользователи смогут писать свои заметки. При отправке этой формы будем выполнять GraphQL-мутацию, создавая заметку в БД.

Для начала создадим в файле src/pages/new.js компонент `NewNote`.

```
import React, { useEffect } from 'react';
import { useMutation, gql } from '@apollo/client';
const NewNote = props => {
  useEffect(() => {
```

```
    // Обновляем заголовок документа
    document.title = 'New Note — Notedly';
  });

  return <div>New note</div>;
};

export default NewNote;
```

Далее настраиваем новый маршрут в файле src/pages/index.js.

```
// Импортируем компонент маршрута NewNote
import NewNote from './new';

// Добавляем приватный маршрут в список маршрутов внутри
<PrivateRoute path="/new" component={NewNote} />
```

Поскольку мы будем не только создавать заметки, но и изменять существующие, нужно создать компонент `NoteForm`, который будет служить в качестве разметки и состояния React для редактирования формы заметки.

Мы создадим новый файл src/components/NoteForm.js. Этот компонент будет состоять из элемента формы, содержащего область текста и необходимый минимум стилей. Его функциональность будет во многом схожа с компонентом `UserForm`.

```
import React, { useState } from 'react';
import styled from 'styled-components';

import Button from './Button';

const Wrapper = styled.div`
  height: 100%;
`;

const Form = styled.form`
  height: 100%;
`;

const TextArea = styled.textarea`
  width: 100%;
  height: 90%;
`;

const NoteForm = props => {
  // Устанавливаем состояние формы по умолчанию
  const [value, setValue] = useState({ content: props.content || '' });

  // Обновляем это состояние при вводе пользователем данных
  const onChange = event => {
    setValue({
      ...value,
```

```
      [event.target.name]: event.target.value
    });
  };

  return (
    <Wrapper>
      <Form
        onSubmit={e => {
          e.preventDefault();
          props.action({
            variables: {
              ...values
            }
          });
        }}
      >
        <TextArea
          required
          type="text"
          name="content"
          placeholder="Note content"
          value={value.content}
          onChange={onChange}
        />
        <Button type="submit">Save</Button>
      </Form>
    </Wrapper>
  );
};

export default NoteForm;
```

Далее нам понадобится сделать ссылку на компонент `NoteForm` из компонента страницы `NewNote`. В src/pages/new.js:

```
import React, { useEffect } from 'react';
import { useMutation, gql } from '@apollo/client';
// Импортируем компонент NoteForm
import NoteForm from '../components/NoteForm';

const NewNote = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'New Note — Notedly';
  });

  return <NoteForm />;
};
export default NewNote;
```

После внесения этих изменений при переходе по ссылке http://localhost:1234/new будет отображаться наша форма (рис. 16.1).

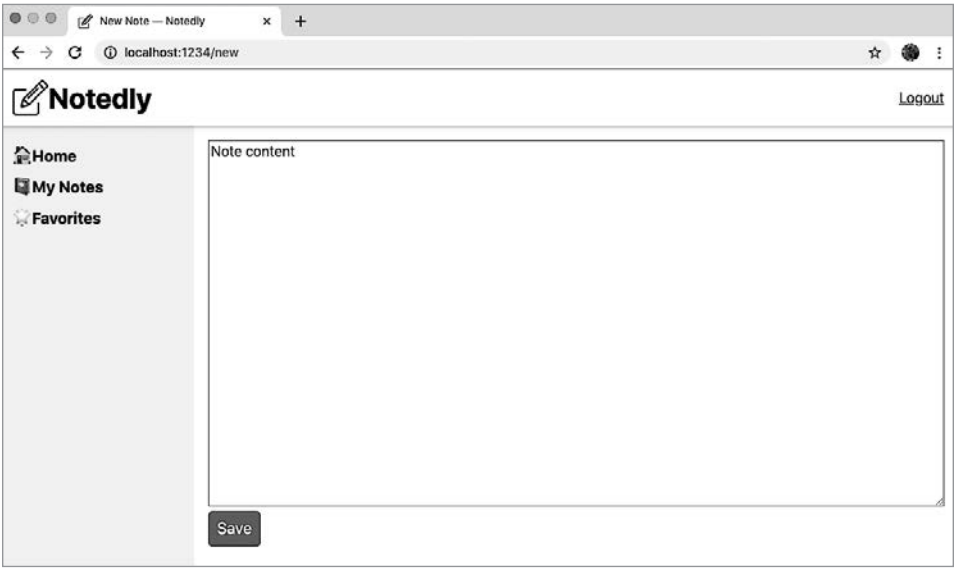

Рис. 16.1. Наш компонент NewNote предоставляет пользователю большое текстовое поле и кнопку Save

Закончив работу с формой, мы можем перейти к написанию мутации для создания новой заметки. В src/pages/new.js:

```
import React, { useEffect } from 'react';
import { useMutation, gql } from '@apollo/client';

import NoteForm from '../components/NoteForm';

// Запрос new note
const NEW_NOTE = gql`
  mutation newNote($content: String!) {
    newNote(content: $content) {
      id
      content
      createdAt
      favoriteCount
      favoritedBy {
        id
        username
      }
      author {
        username
        id
      }
    }
  }
```

```
`;

const NewNote = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'New Note — Notedly';
  });

  const [data, { loading, error }] = useMutation(NEW_NOTE, {
    onCompleted: data => {
      // После завершения перенаправляем пользователя на страницу заметки
      props.history.push(`note/${data.newNote.id}`);
    }
  });

  return (
    <React.Fragment>
      {/* Во время загрузки мутации выдаем сообщение о загрузке */}
      {loading && <p>Loading...</p>}
      {/* В случае сбоя выдаем сообщение об ошибке*/}
      {error && <p>Error saving the note</p>}
      {/* Компонент формы, передающий мутацию данных в качестве prop */}
      <NoteForm action={data} />
    </React.Fragment>
  );
};

export default NewNote;
```

В этом коде при отправке формы мы выполняем мутацию `newNote`. Если мутация выполняется успешно, пользователь перенаправляется на страницу этой заметки. Вы могли обратить внимание, что мутация `newNote` запрашивает достаточно много данных. Это совпадает с данными, запрашиваемыми мутацией `note`, в идеале обновляющей кэш Apollo для быстрого перехода в отдельный компонент заметки.

Как отмечалось ранее, Apollo активно кэширует наши запросы, что помогает ускорить перемещение по приложению. К сожалению, это также означает, что пользователь может перейти на страницу и не увидеть на ней внесенных только что изменений. Мы можем вручную обновить кэш Apollo, но легче будет использовать функцию Apollo `refetchQueries`, чтобы намеренно обновлять кэш при выполнении мутации. Для этого нам нужен доступ к написанным заранее запросам. До сих пор мы включали их в начало файла компонента, но теперь давайте перенесем их в отдельный файл query.js. Создайте его в /src/qql/query.js и добавьте каждый запрос заметки, а также запрос `IS_LOGGED_IN`.

```
import { gql } from '@apollo/client';

const GET_NOTES = gql`
  query noteFeed($cursor: String) {
```

```
    noteFeed(cursor: $cursor) {
      cursor
      hasNextPage
      notes {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;

const GET_NOTE = gql`
  query note($id: ID!) {
    note(id: $id) {
      id
      createdAt
      content
      favoriteCount
      author {
        username
        id
        avatar
      }
    }
  }
`;

const IS_LOGGED_IN = gql`
  {
    isLoggedIn @client
  }
`;

export { GET_NOTES, GET_NOTE, IS_LOGGED_IN };
```

ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ ЗАПРОСОВ И МУТАЦИЙ

По ходу продвижения мы будем хранить все запросы и мутации отдельно от компонентов. Так будет легче использовать их повторно и окажется кстати при создании макетов (https://oreil.ly/qo9uE) в процессе тестирования.

Теперь в файле src/pages/new.js мы можем запросить повторное получение мутацией запроса `GET_NOTES`, импортировав этот запрос и добавив в него опцию `refetchQueries`.

```
// Импортируем запрос
import { GET_NOTES } from '../gql/query';

// Обновляем мутацию в компоненте NewNote
//Все остальное остается прежним

const NewNote = props => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'New Note — Notedly';
  });

  const [data, { loading, error }] = useMutation(NEW_NOTE, {
    // Повторно получаем запрос GET_NOTES, чтобы обновить кэш
    refetchQueries: [{ query: GET_NOTES }],
    onCompleted: data => {
      // В финале перенаправляем пользователя на страницу заметки
      props.history.push(`note/${data.newNote.id}`);
    }
  });

  return (
    <React.Fragment>
      {/* Пока мутация загружается, выдаем сообщение о загрузке */}
      {loading && <p>Loading...</p>}
      {/* В случае сбоя выдаем сообщение об ошибке */}
      {error && <p>Error saving the note</p>}
      {/* Компонент формы, передающий данные мутации в качестве prop */}
      <NoteForm action={data} />
    </React.Fragment>
  );
};
```

Заключительным шагом будет добавление ссылки на страницу /new, чтобы пользователи могли легко переходить туда. Добавьте новый элемент ссылки в файл src/components/Navigation.js следующим образом:

```
<li>
  <Link to="/new">New</Link>
</li>
```

Теперь пользователи смогут переходить на страницу новой заметки, писать ее и сохранять в базе данных.

Чтение заметок пользователей

Сейчас наше приложение может читать как записки по отдельности, так и всю ленту, но мы пока не запрашиваем заметки авторизованных пользователей. Давайте напишем два GraphQL-запроса для генерации ленты обычных или избранных заметок пользователя.

Добавьте запрос GET_MY_NOTES в файл src/qql/query.js и обновите экспорты следующим образом:

```
// Добавляем запрос GET_MY_NOTES
const GET_MY_NOTES = gql`
  query me {
    me {
      id
      username
      notes {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;

// Обновляем для включения GET_MY_NOTES
export { GET_NOTES, GET_NOTE, IS_LOGGED_IN, GET_MY_NOTES };
```

Теперь импортируйте этот запрос в файл src/pages/mynotes.js и отобразите заметки, используя компонент NoteFeed.

```
import React, { useEffect } from 'react';
import { useQuery, gql } from '@apollo/client';

import NoteFeed from '../components/NoteFeed';
import { GET_MY_NOTES } from '../gql/query';

const MyNotes = () => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'My Notes — Notedly';
  });

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

  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return 'Loading...';
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return `Error! ${error.message}`;
  // Если запрос выполнен успешно и содержит заметки, возвращаем их в ленту.
  // Если же запрос выполнен успешно, но заметок в нем нет,
  // выдаем сообщение "No notes yet"
  if (data.me.notes.length !== 0) {
```

```
    return <NoteFeed notes={data.me.notes} />;
  } else {
    return <p>No notes yet</p>;
  }
};

export default MyNotes;
```

Этот процесс можно повторить для создания страницы «Избранное». Сначала в файле src/qql/query.js:

```
// Добавляем запрос GET_MY_FAVORITES
const GET_MY_FAVORITES = gql`
  query me {
    me {
      id
      username
      favorites {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;

// Обновляем для включения GET_MY_FAVORITES
export { GET_NOTES, GET_NOTE, IS_LOGGED_IN, GET_MY_NOTES, GET_MY_FAVORITES };
```

Теперь в файле src/pages/favorites.js:

```
import React, { useEffect } from 'react';
import { useQuery, gql } from '@apollo/client';

import NoteFeed from '../components/NoteFeed';
// Импортируем запрос
import { GET_MY_FAVORITES } from '../gql/query';

const Favorites = () => {
  useEffect(() => {
    // Обновляем заголовок документа
    document.title = 'Favorites — Notedly';
  });

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

  // Если данные загружаются, выдаем сообщение о загрузке
```

```
  if (loading) return 'Loading...';
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return `Error! ${error.message}`;
  // Если запрос выполнен успешно и содержит заметки, возвращаем их в ленту
  // Если же запрос выполнен успешно, но заметок не содержит,
  // выдаем сообщение "No favorites yet"
  if (data.me.favorites.length !== 0) {
    return <NoteFeed notes={data.me.favorites} />;
  } else {
    return <p>No favorites yet</p>;
  }
};

export default Favorites;
```

В конце давайте обновим файл src/pages/new.js, чтобы повторно получать запрос `GET_MY_NOTES`, обеспечив тем самым обновление кэшированного списка заметок при создании новой. В файле src/pages/new.js сначала обновите инструкцию импорта GraphQL-запроса:

```
import { GET_MY_NOTES, GET_NOTES } from '../gql/query';
```

А затем обновите мутацию:

```
const [data, { loading, error }] = useMutation(NEW_NOTE, {
  // Повторно получаем запросы GET_NOTES и GET_MY_NOTES, чтобы обновить кэш
  refetchQueries: [{ query: GET_MY_NOTES }, { query: GET_NOTES }],
  onCompleted: data => {
    // В конце перенаправляем пользователя на страницу заметки
    props.history.push(`note/${data.newNote.id}`);
  }
});
```

После внесения этих изменений в приложении стали доступны всевозможные операции чтения.

Изменение заметок

Сейчас пользователи после написания заметки не имеют возможности ее изменить. Чтобы это исправить, нам нужно добавить в приложение возможность редактирования заметок. В нашем GraphQL API есть мутация `updateNote`, которая принимает ID заметки и ее содержимое в качестве параметров. Если заметка существует в базе данных, мутация обновит хранящееся там содержимое на то, что отправлено в мутации.

В приложении можно создать маршрут /edit/NOTE_ID, который будет помещать содержимое существующей заметки в форму `textArea`. При нажатии пользователем Save мы будем отправлять эту форму и выполнять мутацию `updateNote`.

Давайте создадим новый маршрут, по которому будут редактироваться заметки. Для начала можно сделать копию страницы src/pages/note.js и назвать ее edit.js. Пока что эта страница будет просто отображать заметку.

В src/pages/edit.js:

```
import React from 'react';
import { useQuery, useMutation, gql } from '@apollo/client';

// Импортируем компонент Note
import Note from '../components/Note';
// Импортируем запрос GET_NOTE
import { GET_NOTE } from '../gql/query';

const EditNote = props => {
  // Сохраняем id, полученный из url, в виде переменной
  const id = props.match.params.id;
  // Определяем запрос заметки
  const { loading, error, data } = useQuery(GET_NOTE, { variables: { id } });

  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return 'Loading...';
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <p>Error! Note not found</p>;
  // В случае успеха передаем данные в компонент note
  return <Note note={data.note} />;
};

export default EditNote;
```

Теперь можно открыть доступ к этой странице, добавив ее в список маршрутов файла src/pages/index.js:

```
// Импортируем компонент страницы edit
import EditNote from './edit';

// Добавляем новый приватный маршрут, принимающий параметр :id
<PrivateRoute path="/edit/:id" component={EditNote} />
```

Если теперь вы перейдете на страницу заметки по адресу /note/ID и замените ее на /edit/ID, то увидите саму заметку. Давайте внесем изменения, чтобы вместо этого отображалось содержимое, представленное в `textarea` формы.

В файле src/pages/edit.js удалите инструкцию импорта компонента `Note`, заменив его на компонент `NoteForm`:

```
// Импортируем компонент NoteForm
import NoteForm from '../components/NoteForm';
```

Теперь можно обновить компонент `EditNote`, чтобы использовать форму редактирования. Можно передать содержимое заметки в компонент формы, используя свойство `content`. Наша GraphQL-мутация принимает обновления только от

автора заметки, но будет лучше, если и показываться она будет только автору, чтобы не вводить в замешательство других пользователей.

Сначала добавьте в файл src/qql/query.js новый запрос для получения текущего пользователя, его ID и списка ID избранных заметок.

```
// Добавляем в запросы GET_ME
const GET_ME = gql`
  query me {
    me {
      id
      favorites {
        id
      }
    }
  }
`;

// Обновляем для включения GET_ME
export {
  GET_NOTES,
  GET_NOTE,
  GET_MY_NOTES,
  GET_MY_FAVORITES,
  GET_ME,
  IS_LOGGED_IN
};
```

Импортируйте запрос `GET_ME` в файл src/pages/edit.js и добавьте проверку пользователя.

```
import React from 'react';
import { useMutation, useQuery } from '@apollo/client';

// Импортируем компонент NoteForm
import NoteForm from '../components/NoteForm';
import { GET_NOTE, GET_ME } from '../gql/query';
import { EDIT_NOTE } from '../gql/mutation';

const EditNote = props => {
  // Сохраняем id, полученный из url, в виде переменной
  const id = props.match.params.id;
  // Определяем запрос заметки
  const { loading, error, data } = useQuery(GET_NOTE, { variables: { id } });
  // Получаем информацию о текущем пользователе
  const { data: userdata } = useQuery(GET_ME);
  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return 'Loading...';
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <p>Error! Note not found</p>;
  // Если текущий пользователь не соответствует автору заметки,
  // возвращаем соответствующее сообщение
```

```
  if (userdata.me.id !== data.note.author.id) {
    return <p>You do not have access to edit this note</p>;
  }
  // Передаем данные в компонент формы
  return <NoteForm content={data.note.content} />;
};
```

Теперь у нас есть возможность редактировать заметки в форме, но нажатие кнопки пока не ведет к сохранению изменений. Давайте напишем мутацию `updateNote`. Для этого аналогично файлу запросов мы создадим файл для хранения мутаций. Добавьте в src/qql/mutation.js следующее:

```
import { gql } from '@apollo/client';

const EDIT_NOTE = gql`
  mutation updateNote($id: ID!, $content: String!) {
    updateNote(id: $id, content: $content) {
      id
      content
      createdAt
      favoriteCount
      favoritedBy {
        id
        username
      }
      author {
        username
        id
      }
    }
  }
`;

export { EDIT_NOTE };
```

Написав мутацию, мы можем импортировать ее и обновить код компонента, чтобы вызывать ее при нажатии кнопки. Для этого мы добавим хук `useMutation`. После завершения мутации мы будем перенаправлять пользователя на страницу заметки.

```
// Импортируем мутацию
import { EDIT_NOTE } from '../gql/mutation';

const EditNote = props => {
  // Сохраняем id, полученный из url, в виде переменной
  const id = props.match.params.id;
  // Определяем запрос заметки
  const { loading, error, data } = useQuery(GET_NOTE, { variables: { id } });
  // Получаем информацию о текущем пользователе
  const { data: userdata } = useQuery(GET_ME);
  // Определяем мутацию
  const [editNote] = useMutation(EDIT_NOTE, {
```

```
    variables: {
      id
    },
    onCompleted: () => {
      props.history.push(`/note/${id}`);
    }
  });

  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return 'Loading...';
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <p>Error!</p>;
  // Если текущий пользователь не соответствует автору заметки
  if (userdata.me.id !== data.note.author.id) {
    return <p>You do not have access to edit this note</p>;
  }

  // передаем данные и мутацию в компонент формы
  return <NoteForm content={data.note.content} action={editNote} />;
};

export default EditNote;
```

Наконец, мы хотим отображать для пользователя ссылку Edit, но только в случае, когда он является автором просматриваемой заметки. Для этого нам понадобится выполнять проверку совпадения ID текущего пользователя с ID автора заметки. Для реализации такого поведения мы задействуем несколько компонентов.

Теперь можно было бы реализовать эту функциональность напрямую в компоненте `Note`, но давайте лучше создадим компонент src/components/NoteUser.js конкретно для действий авторизованных пользователей. В этом компоненте мы будем выполнять GraphQL-запрос для получения ID текущего пользователя и предоставлять переходную ссылку для редактирования страницы. С этой информацией мы можем начать с добавления необходимых библиотек и настройки компонента React. В этот компонент мы включим ссылку для редактирования, которая будет направлять пользователей на страницу редактирования заметки. Пока что пользователь будет видеть эту ссылку независимо от авторства.

Обновите файл src/components/NoteUser.js следующим образом:

```
import React from 'react';
import { useQuery, gql } from '@apollo/client';
import { Link } from 'react-router-dom';

const NoteUser = props => {
  return <Link to={`/edit/${props.note.id}`}>Edit</Link>;
};

export default NoteUser;
```

Затем мы обновим компонент `Note`, чтобы он выполнял запрос локального состояния `isLoggedIn`. После этого можно условно отображать компонент `NoteUser`, ориентируясь на состояние авторизации пользователя.

Давайте сначала импортируем компонент `UserNote`, а также GraphQL-библиотеки для выполнения запроса. Добавьте в начало файла src/components/Note.js следующее:

```
import { useQuery } from '@apollo/client';

// Импортируем компоненты UI авторизованного пользователя
import NoteUser from './NoteUser';
// Импортируем локальный запрос IS_LOGGED_IN
import { IS_LOGGED_IN } from '../gql/query';
```

Теперь мы можем обновить JSX-компонент, чтобы проверять состояние авторизации. Если пользователь авторизован, мы будем отображать компонент `NoteUser`, в противном же случае будем показывать число избранных заметок.

```
const Note = ({ note }) => {
  const { loading, error, data } = useQuery(IS_LOGGED_IN);
  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <p>Error!</p>;

  return (
    <StyledNote>
      <MetaData>
        <MetaInfo>
          <img
            src={note.author.avatar}
            alt={`${note.author.username} avatar`}
            height="50px"
          />
        </MetaInfo>
        <MetaInfo>
          <em>by</em> {note.author.username} <br />
          {format(note.createdAt, 'MMM Do YYYY')}
        </MetaInfo>
        {data.isLoggedIn ? (
          <UserActions>
            <NoteUser note={note} />
          </UserActions>
        ) : (
          <UserActions>
            <em>Favorites:</em> {note.favoriteCount}
          </UserActions>
        )}
      </MetaData>
      <ReactMarkdown source={note.content} />
    </StyledNote>
  );
};
```

НЕСАНКЦИОНИРОВАННОЕ РЕДАКТИРОВАНИЕ

Хотя мы будем прятать ссылку на редактирование в UI, пользователи по-прежнему смогут переходить на вкладку редактирования заметки, не будучи ее владельцем. К счастью, наш GraphQL API спроектирован так, что изменять содержимое заметки сможет только ее создатель. Хотя мы и не будем реализовывать это в данной книге, хорошим дополнительным шагом было бы обновить компонент src/pages/edit.js, чтобы он перенаправлял пользователей, не являющихся авторами заметки.

После внесения всех этих изменений пользователи смогут видеть ссылку для редактирования в верхней части каждой заметки. Переходя по ней, они будут попадать в форму для редактирования, независимо от того, кто является владельцем заметки. Давайте это исправим, обновив компонент `NoteUser`, чтобы запрашивать ID пользователя и отображать ссылку для редактирования только в случае совпадения этого ID с ID автора заметки.

Сначала добавьте в файл src/components/NoteUser.js следующее:

```
import React from 'react';
import { useQuery } from '@apollo/client';
import { Link } from 'react-router-dom';

// Импортируем запрос GET_ME
import { GET_ME } from '../gql/query';

const NoteUser = props => {
  const { loading, error, data } = useQuery(GET_ME);
  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <p>Error!</p>;
  return (
    <React.Fragment>
      Favorites: {props.note.favoriteCount}
      <br />
      {data.me.id === props.note.author.id && (
        <React.Fragment>
          <Link to={`/edit/${props.note.id}`}>Edit</Link>
        </React.Fragment>
      )}
    </React.Fragment>
  );
};

export default NoteUser;
```

После внесения этих изменений ссылку в UI для редактирования заметки будет видеть только ее автор (рис. 16.2).

Рис. 16.2. Ссылку для редактирования будет видеть только автор заметки

Удаление заметок

Нашему CRUD-приложению по-прежнему недостает возможности удаления заметок. Мы можем написать UI-компонент кнопки, который при срабатывании будет выполнять GraphQL-мутацию, удаляя заметку. Давайте начнем с создания нового компонента src/components/DeleteNote.js. Поскольку мы будем выполнять перенаправление в немаршрутизируемом компоненте, нам снова придется использовать компонент верхнего уровня `withRouter`.

```
import React from 'react';
import { useMutation } from '@apollo/client';
import { withRouter } from 'react-router-dom';

import ButtonAsLink from './ButtonAsLink';

const DeleteNote = props => {
  return <ButtonAsLink>Delete Note</ButtonAsLink>;
};

export default withRouter(DeleteNote);
```

Теперь можно написать мутацию. В нашем GraphQL API есть мутация `deleteNote`, которая возвращает логическое значение `true`, если заметка удалена. При завершении этой мутации мы будем перенаправлять пользователя на страницу приложения /mynotes.

Сначала пропишите мутацию в src/qql/mutation.js:

```
const DELETE_NOTE = gql`
  mutation deleteNote($id: ID!) {
    deleteNote(id: $id)
  }
`;

// Добавляем DELETE_NOTE
export { EDIT_NOTE, DELETE_NOTE };
```

Теперь добавьте в файл src/components/DeleteNote следующее:

```
import React from 'react';
import { useMutation } from '@apollo/client';
import { withRouter } from 'react-router-dom';

import ButtonAsLink from './ButtonAsLink';
// Импортируем мутацию DELETE_NOTE
import { DELETE_NOTE } from '../gql/mutation';
// Импортируем запросы для их повторного получения после удаления заметки
import { GET_MY_NOTES, GET_NOTES } from '../gql/query';

const DeleteNote = props => {
  const [deleteNote] = useMutation(DELETE_NOTE, {
    variables: {
      id: props.noteId
    },
    // Повторно получаем запросы списка заметок, чтобы обновить кэш
    refetchQueries: [{ query: GET_MY_NOTES, GET_NOTES }],
    onCompleted: data => {
      // Перенаправляем пользователя на страницу "my notes"
      props.history.push('/mynotes');
    }
  });

  return <ButtonAsLink onClick={deleteNote}>Delete Note</ButtonAsLink>;
};

export default withRouter(DeleteNote);
```

Теперь можно импортировать новый компонент `DeleteNote` в файл src/components/NoteUser.js, отображая его только автору заметки.

```
import React from 'react';
import { useQuery } from '@apollo/client';
import { Link } from 'react-router-dom';

import { GET_ME } from '../gql/query';
// Импортируем компонент DeleteNote
import DeleteNote from './DeleteNote';
```

```
const NoteUser = props => {
  const { loading, error, data } = useQuery(GET_ME);
  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return <p>Loading...</p>;
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <p>Error!</p>;

  return (
    <React.Fragment>
      Favorites: {props.note.favoriteCount} <br />
      {data.me.id === props.note.author.id && (
        <React.Fragment>
          <Link to={`/edit/${props.note.id}`}>Edit</Link> <br />
          <DeleteNote noteId={props.note.id} />
        </React.Fragment>
      )}
    </React.Fragment>
  );
};

export default NoteUser;
```

Написав эту мутацию, мы предоставили авторизованным пользователям возможность удалять заметки нажатием кнопки.

Добавление в «Избранное»

Теперь нашему приложению недостает последнего элемента функциональности — возможности добавлять и удалять избранные заметки. Давайте для создания и интеграции этой функциональности в приложение последуем тому же шаблону создания компонента. Сначала создайте новый компонент src/components/FavoriteNote.js:

```
import React, { useState } from 'react';
import { useMutation } from '@apollo/client';

import ButtonAsLink from './ButtonAsLink';

const FavoriteNote = props => {
  return <ButtonAsLink>Add to favorites</ButtonAsLink>;
};

export default FavoriteNote;
```

Прежде чем добавлять какую-либо функциональность, давайте включим этот компонент в src/components/NoteUser.js. Сначала импортируйте сам компонент:

```
import FavoriteNote from './FavoriteNote';
```

Теперь включите в JSX-ссылку на него. Вы можете вспомнить, что при написании запроса `GET_ME` мы включили список ID избранных заметок, который также пригодится здесь.

```
return (
  <React.Fragment>
    <FavoriteNote
      me={data.me}
      noteId={props.note.id}
      favoriteCount={props.note.favoriteCount}
    />
    <br />
    {data.me.id === props.note.author.id && (
      <React.Fragment>
        <Link to={`/edit/${props.note.id}`}>Edit</Link> <br />
        <DeleteNote noteId={props.note.id} />
      </React.Fragment>
    )}
  </React.Fragment>
);
```

Вы заметите, что мы передаем в компонент `FavoriteNote` три свойства. Первое — это данные `me`, которые будут включать ID текущего пользователя и список заметок, отмеченных им как избранные. Второе — это `noteID` текущей заметки. Третье — это `favoriteCount`, представляющий общее число избранных заметок пользователя.

Теперь мы можем вернуться к нашему файлу src/components/FavoriteNote.js. В нем в виде состояния мы будем хранить текущее число избранных заметок и проверять, присутствует ли ID текущей заметки в их списке. Будем изменять видимый пользователю текст на основе состояния его избранных заметок. При нажатии пользователем кнопки она будет вызывать мутацию `toggleFavorite`, которая либо добавит заметку в список избранных, либо удалит ее оттуда. Давайте начнем с обновления компонента, чтобы использовать состояние для контроля функции нажатия кнопки.

```
const FavoriteNote = props => {
  // Сохраняем число избранных заметок пользователя как состояние
  const [count, setCount] = useState(props.favoriteCount);

  // Если пользователь отметил заметку как избранную, сохраняем
  // это как состояние
  const [favorited, setFavorited] = useState(
    // Проверяем, присутствует ли заметка в списке избранных
    props.me.favorites.filter(note => note.id === props.noteId).length > 0
  );

  return (
    <React.Fragment>
      {favorited ? (
```

```
        <ButtonAsLink
          onClick={() => {
            setFavorited(false);
            setCount(count - 1);
          }}
        >
          Remove Favorite
        </ButtonAsLink>
      ) : (
        <ButtonAsLink
          onClick={() => {
            setFavorited(true);
            setCount(count + 1);
          }}
        >
          Add Favorite
        </ButtonAsLink>
      )}
      : {count}
    </React.Fragment>
  );
};
```

Внеся все эти изменения, мы будем обновлять состояние при нажатии пользователем кнопки, но вызов мутации при этом пока еще не осуществляется. Давайте завершим этот компонент, написав мутацию и добавив ее в него. Итоговый результат показан на рис. 16.3.

В src/gql/mutation.js:

```
// Добавляем мутацию TOGGLE_FAVORITE
const TOGGLE_FAVORITE = gql`
  mutation toggleFavorite($id: ID!) {
    toggleFavorite(id: $id) {
      id
      favoriteCount
    }
  }
`;

// Добавляем TOGGLE_FAVORITE
export { EDIT_NOTE, DELETE_NOTE, TOGGLE_FAVORITE };
```

В src/components/FavoriteNote.js:

```
import React, { useState } from 'react';
import { useMutation } from '@apollo/client';

import ButtonAsLink from './ButtonAsLink';
// Импортируем мутацию TOGGLE_FAVORITE
import { TOGGLE_FAVORITE } from '../gql/mutation';
// Добавляем запрос GET_MY_FAVORITES для его повторного получения
import { GET_MY_FAVORITES } from '../gql/query';
```

```
  const FavoriteNote = props => {
    // Сохраняем число избранных заметок как состояние
    const [count, setCount] = useState(props.favoriteCount);

    // Если пользователь отметил заметку как избранную, сохраняем это как
состояние
    const [favorited, setFavorited] = useState(
      // Проверяем, присутствует ли заметка в списке избранных
      props.me.favorites.filter(note => note.id === props.noteId).length > 0
    );

    //Хук мутации toggleFavorite
    const [toggleFavorite] = useMutation(TOGGLE_FAVORITE, {
      variables: {
        id: props.noteId
      },
      // Повторно получаем запрос GET_MY_FAVORITES для обновления кэша
      refetchQueries: [{ query: GET_MY_FAVORITES }]
    });

    // Если пользователь добавил заметку в избранное, отображаем
    // вариант ее удаления из списка.
    // В противном случае отображаем вариант ее добавления
    return (
      <React.Fragment>
        {favorited ? (
          <ButtonAsLink
            onClick={() => {
              toggleFavorite();
              setFavorited(false);
              setCount(count - 1);
            }}
          >
            Remove Favorite
          </ButtonAsLink>
        ) : (
          <ButtonAsLink
            onClick={() => {
              toggleFavorite();
              setFavorited(true);
              setCount(count + 1);
            }}
          >
            Add Favorite
          </ButtonAsLink>
        )}
        : {count}
      </React.Fragment>
    );
  };

  export default FavoriteNote;
```

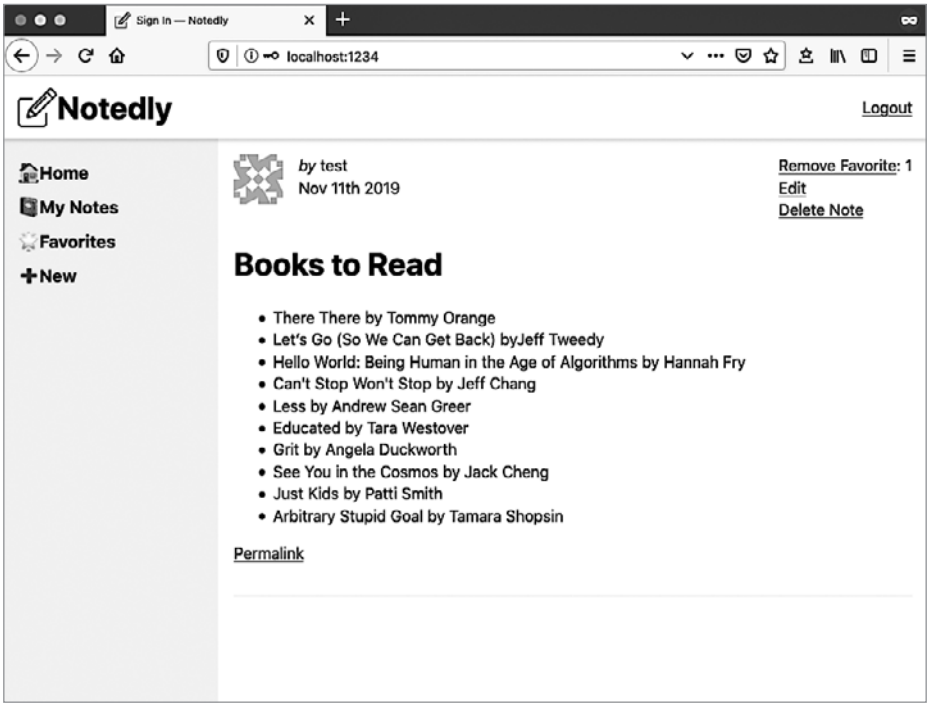

Рис. 16.3. Авторизованный пользователь сможет создавать, читать, редактировать и удалять заметки

Итоги

В этой главе мы превратили наш сайт в полноценное CRUD-приложение. Теперь мы можем реализовать GraphQL-запросы и мутации на основе состояния авторизации пользователя. Навык построения UI, интегрирующих CRUD-функциональность для пользователей, станет прочным фундаментом для создания всевозможных веб-приложений. С этой функциональностью наше приложение достигло уровня MVP (minimum viable product — «минимальный жизнеспособный продукт»). В следующей главе мы развернем это приложение на веб-сервере.

ГЛАВА 17

Развертывание приложения

Когда я только начал заниматься веб-разработкой профессионально, термин «развертывание» означал выгрузку файлов с локальной машины на веб-сервер через FTP-клиент. Не существовало этапов сборки или конвейеров, в связи с чем сырые файлы на компьютере совпадали с находящимися на конечном сервере. Если что-то шло не так, я либо неистово стремился исправить неполадку, либо делал откат изменений, замещая их копиями старых файлов. Такой подход в стиле «Дикого Запада» в свое время вполне работал, но вел к частым падениям сайта и неожиданным проблемам.

В современном мире веб-разработки потребности локальной среды и веб-серверов сильно отличаются. Я хочу видеть на своем компьютере мгновенные изменения при обновлении файла и иметь несжатые файлы для отладки. Изменения на моем веб-сервере я ожидаю видеть только при их развертывании, предпочитая при этом файлы небольшого размера. В этой главе мы рассмотрим один из способов развертывания статических приложений в интернете.

Статические сайты

Браузер считывает HTML, CSS и JavaScript, чтобы сгенерировать страницы, с которыми мы в итоге взаимодействуем. В отличие от таких фреймворков, как Express, Rails и Django, генерирующих разметку для страницы на стороне сервера во время запроса, статические сайты являются простой коллекцией HTML, CSS и JavaScript, хранящихся на сервере. В итоге их сложность может варьироваться от одного HTML-файла с разметкой до сложного процесса сборки фронтенда, который компилирует языки шаблонов, множество JS-файлов и препроцессоры CSS. Как бы то ни было, в итоге статические сайты представляют собой коллекцию этих трех типов файлов.

Наше приложение является статическим. Оно содержит разметку, CSS и JS-код. Используемый нами инструмент сборки Parcel (https://parceljs.org) компилирует эти компоненты в понятные браузеру файлы. В локальной разработке мы запускаем сервер, и далее эти файлы мгновенно обновляются благодаря функции

горячей смены модулей. Заглянув в файл package.json, вы увидите, что я включил два скрипта deploy:

```
"scripts": {
  "deploy:src": "parcel build src/index.html --public-url ./",
  "deploy:final": "parcel build final/index.html --public-url ./"
}
```

Для сборки приложения откройте терминал, используйте **cd** для перемещения в корень директории web, содержащей проект, а затем выполните команду **build**.

```
# Если вы не находитесь в директории web, перейдите в нее с помощью команды cd
$ cd Projects/notedly/веб
# Выполните сборку файлов из директории src
$ npm run deploy:src
```

Если вы следовали инструкциям книги и вели разработку приложения в директории src, то запуск `npm run deploy: src` в терминале выполнит сборку приложения на основе вашего кода. Если же вы предпочтете использовать последнюю версию приложения, поставляемую с примером кода, то используйте команду `npm run deploy:final` для сборки приложения из директории final.

В оставшейся части главы я продемонстрирую один из способов развертывания статически собранного приложения, но эти файлы можно разместить в любом месте, где поддерживается HTML, начиная от провайдера веб-хостинга и заканчивая Raspberry Pi, оставленного на вашем столе. Несмотря на то что рассматриваемый процесс имеет множество ощутимых преимуществ, развертывание может быть упрощено до указания в .env-файле удаленного API, выполнения сценария сборки и загрузки файлов.

СЕРВЕРНЫЕ REACT-ПРИЛОЖЕНИЯ

Хотя мы и создаем веб-приложение как статическое, отображение JSX на сервере также возможно. Эта техника, которую часто называют «универсальный JavaScript», может дать несколько преимуществ, включая прирост производительности, резервные JS-элементы на клиентской стороне и улучшение SEO-показателей. Такие фреймворки, как Next.js (https://nextjs.org/), ставят цель упростить эту настройку. Несмотря на то что в этой книге мы не затрагиваем серверные JS-приложения, я рекомендую изучить этот подход после того, как вы освоите разработку клиентских JS-приложений.

Конвейер развертывания

Для развертывания нашего приложения мы используем простой конвейер, который позволит автоматически внедрять изменения в кодовую базу. Для этого конвейера мы будем использовать два сервиса. Первым будет репозиторий исходного кода на GitHub (https:// github.com). Вторым будет платформа веб-

хостинга Netlify (https://www.netlify.com). Я выбрал Netlify за ее обширный и при этом легкий в использовании набор функций для развертывания, а также потому, что она ориентирована на статические и бессерверные приложения.

Нашей целью будет сделать так, чтобы любой коммит в `master`-ветку приложения автоматически развертывался на веб-хосте. Визуально этот процесс представлен на рис. 17.1.

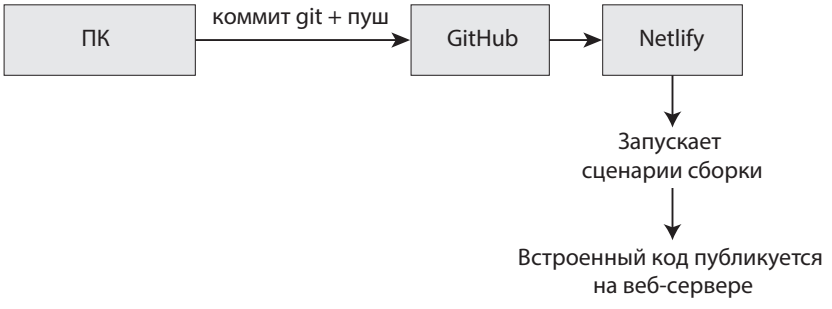

Рис. 17.1. Процесс развертывания

Хостинг исходного кода с помощью Git

Первым шагом процесса будет настройка репозитория исходного кода. Если вам этот этап хорошо знаком, можете смело перейти к следующему шагу. Как уже отмечалось, мы будем использовать GitHub (https://github.com/), но этот процесс можно сконфигурировать и с помощью других публичных Git-хостов, например GitLab (https://about.gitlab.com) или Bitbucket (https:// bitbucket.org).

РЕПОЗИТОРИИ GITHUB

Мы будем создавать новый репозиторий на GitHub, но при желании вы можете использовать официальный образец кода, доступный по ссылке https://github.com/javascripteverywhere/web, создав форк к своему аккаунту на GitHub.

Сначала перейдите на GitHub и создайте учетную запись или авторизуйтесь в существующей. Затем нажмите кнопку New Repository, введите имя и кликните на Create Repository (рис. 17.2).

Теперь в терминале перейдите в директорию приложения, установите в качестве источника Git новый репозиторий GitHub и отправьте код. Так как мы обновляем существующий репозиторий Git, наши инструкции будут несколько отличаться от предлагаемых GitHub.

```
# Перейдите в директорию web
cd Projects/notedly/web
```

```
# Укажите свой репозиторий в качестве удаленного источника GitHub
git remote set-url origin git://YOUR.GIT.URL
# Отправьте код в новый репозиторий GitHub
git push -u origin master
```

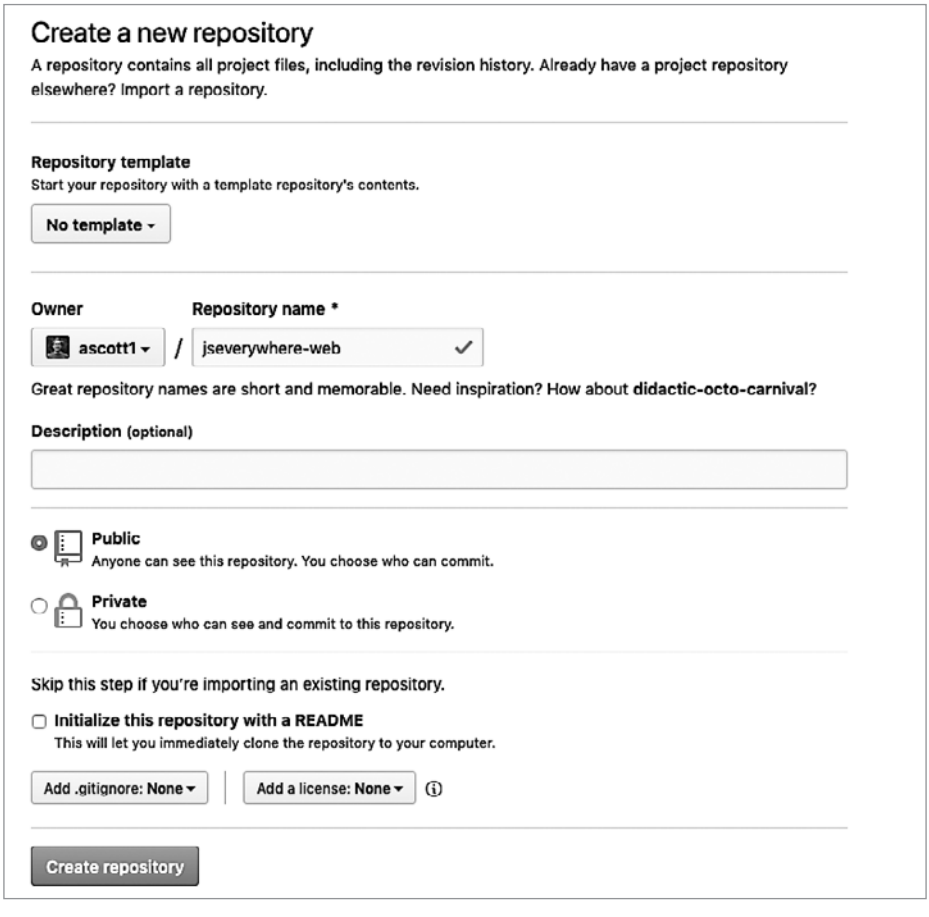

Рис. 17.2. Страница нового репозитория на GitHub

Если теперь вы перейдете по ссылке https://github.com/<ваше_имя_пользователя>/<ваше_имя_репозитория>, то увидите исходный код приложения.

Развертывание с помощью Netlify

Разместив код в удаленном Git-репозитории, мы можем настроить веб-хост Netlify для сборки и развертывания кода. Сначала перейдите на сайт netlify.com

и создайте аккаунт. После этого кликните по New site from Git и проследуйте инструкциям по настройке развертывания сайта.

1. Укажите в качестве Git-поставщика GitHub, и ваш GitHub-аккаунт будет подключен и авторизован.
2. Выберите репозиторий с исходным кодом.
3. Настройте конфигурацию сборки.

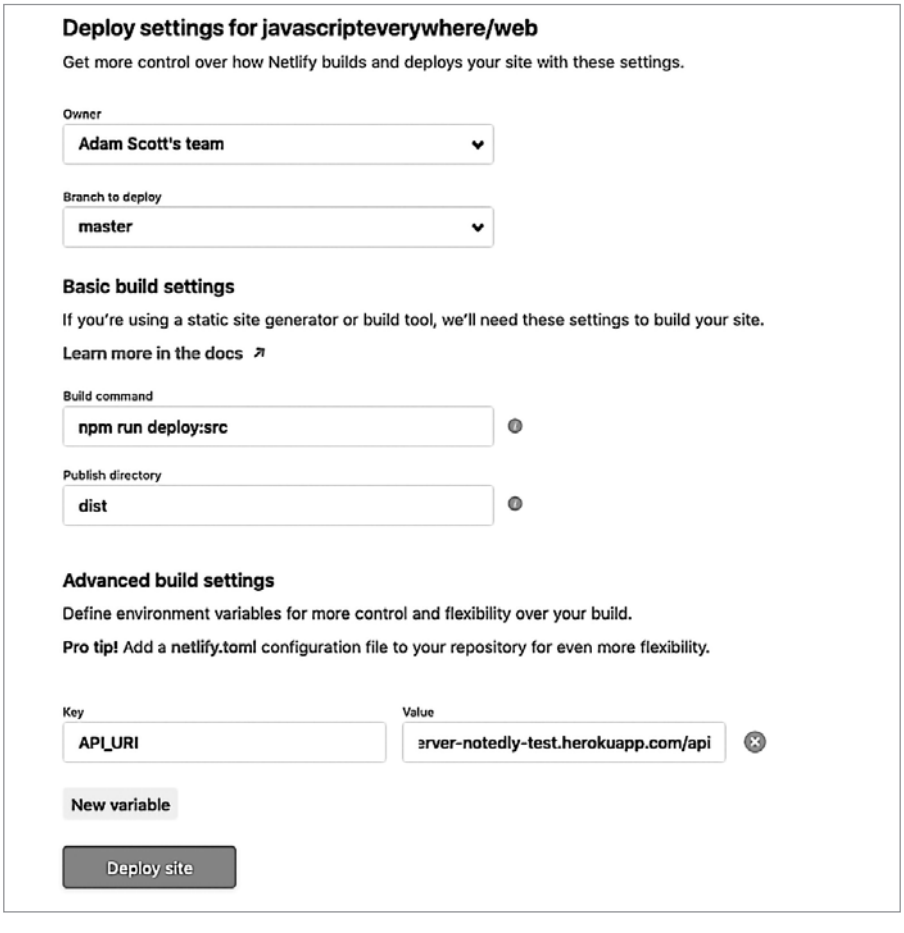

Рис. 17.3. С помощью Netlify можно настраивать процесс сборки и переменные среды

К нашим настройкам сборки добавьте следующее (рис. 17.3):

1. Команду сборки: `npm run deploy:src` (или `npm run deploy:final`, если развертываете финальный образец кода).

2. Директорию публикации dist.
3. В разделе Advanced settings выберите New variable и добавьте имя переменной `API_URI`, определив ее значение как https://<имя_вашего_api>.herokuapp.com/api. Это будет URL приложения API, которое мы развернули в Heroku.

Закончив настройку приложения, нажмите кнопку Deploy site. Спустя несколько минут ваше приложение будет выполняться по URL, предоставленному Netlify. Теперь при каждой передаче изменений в репозиторий на GitHub наш сайт будет развертываться автоматически.

МЕДЛЕННАЯ НАЧАЛЬНАЯ ЗАГРУЗКА

Развернутое приложение будет загружать данные из ранее развернутого нами Heroku API. При использовании бесплатного тарифа Heroku контейнеры приложения входят в спящий режим спустя час простоя. Если вы не используете API какое-то время, начальная загрузка данных будет медленной, пока контейнер не войдет в рабочий режим.

Итоги

В пройденной главе мы развернули статическое веб-приложение. Для этого мы использовали функции конвейера развертывания Netlify, чтобы отслеживать изменения в Git-репозитории, выполнять процессы сборки и хранить переменные среды. Используя это как основу, мы можем полноценно выпускать публичные веб-приложения.

ГЛАВА 18

Создание десктопных приложений с помощью Electron

Мое знакомство с персональными компьютерами произошло в школьном кабинете, где стояло множество машин Apple II. Раз в неделю нас с одноклассниками приводили в это помещение, выдавали несколько дискет и приблизительный набор инструкций по загрузке приложения (обычно Oregon Trail). Я мало что помню из этих занятий, кроме ощущения полной привязанности к маленькому мирку, который я теперь мог контролировать. Персональные компьютеры прошли долгий путь развития, начиная с середины 80-х, но при выполнении многих задач мы по-прежнему полагаемся именно на десктопные приложения.

В течение своего привычного рабочего дня я могу обращаться к почтовому клиенту, текстовому редактору, чат-клиенту, электронным таблицам, стриминговому музыкальному сервису и ряду других десктопных приложений. Зачастую у них есть веб-эквивалент, но удобство и интеграция десктопного приложения дает ряд преимуществ в плане использования. Тем не менее на протяжении долгих лет создание таких приложений казалось недосягаемым. К счастью, сегодня мы можем использовать веб-технологии для создания полноценных десктопных приложений, не затрачивая много времени на обучение.

Что мы создаем

На протяжении нескольких следующих глав мы будем создавать десктопный клиент для нашего приложения социальных заметок, Notedly. Целью будет использовать JavaScript и веб-технологии для разработки приложения, которое пользователь сможет скачать и установить на компьютер. Пока что это приложение будет простой реализацией, которая обернет наше веб-приложение в десктопную оболочку. С помощью такой разработки продукта мы сможем быстро поставить его заинтересованным людям и в то же время позднее предложить кастомизированное приложение для пользователей настольных компьютеров.

Как мы будем это создавать

Для сборки приложения мы будем использовать Electron (https://electronjs.org/), фреймворк с открытым исходным кодом, предназначенный для создания кроссплатформенных десктопных приложений с помощью веб-технологий. Работает он на основе Node.js и браузерного движка Chromium. Это означает, что как у разработчиков у нас есть доступ к миру браузера, Node.js и возможностям операционной системы, которые обычно недоступны в веб-среде. Изначально Electron был разработан GitHub для текстового редактора Atom (https://atom.io), но с тех пор он также служит платформой для больших и малых приложений, включая Slack, VS Code, Discord и WordPress Desktop.

Начало

Прежде чем перейти к разработке, нужно сделать копию стартовых файлов проекта на компьютер. Исходный код проекта содержит все сценарии и ссылки на сторонние библиотеки, которые нам понадобятся при создании приложения. Чтобы клонировать код на локальную машину, откройте терминал, перейдите в директорию, где хранятся проекты, и выполните команду **`git clone`** для репозитория проекта. Если вы также следовали процессу разработки в главах, посвященных приложениям API и веб-составляющей, у вас уже должна быть директория notedly, где хранится организованный проект.

```
$ cd Projects
$ # Если у вас еще нет директории notedly, введите команду `mkdir notedly`
$ cd notedly
$ git clone git@github.com:javascripteverywhere/desktop.git
$ cd desktop
$ npm install
```

УСТАНОВКА СТОРОННИХ ЗАВИСИМОСТЕЙ

Сделав копию стартового кода книги и выполнив команду `npm install` в директории, вы избегаете необходимости повторно выполнять эту команду для каждой отдельной сторонней зависимости.

Структура кода:

/src

Директория для ведения разработки по ходу изучения книги.

/solutions

Директория, содержащая решения для каждой главы. Если у вас возникнут сложности, можете обратиться к ней в поиске ответа.

/final

Директория для финальной версии рабочего проекта.

Создав каталог проекта и установив зависимости, мы подготовились к началу самого процесса разработки.

Наше первое приложение на Electron

Скопировав репозиторий на компьютер, можно переходить к разработке нашего первого Electron-приложения. Если вы заглянете в директорию src, то увидите несколько файлов. Файл index.html содержит простую HTML-разметку. Пока что этот файл будет служить в качестве «процесса отрисовки» Electron, то есть будет представлять веб-страницу, отображаемую в окне нашего приложения.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Notedly Desktop</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

В файле index.js мы будем настраивать приложение. Он будет содержать так называемый главный процесс, задача которого заключается в создании в Electron экземпляра `BrowserWindow`, используемого в качестве оболочки приложения.

INDEX.JS ВМЕСТО MAIN.JS

Несмотря на то что я назвал этот файл index.js, чтобы придерживаться единого шаблона, используемого в остальных образцах наших приложений, в разработке на Electron файл главного процесса принято называть main.js.

Давайте настроим главный процесс на отображение окна браузера, содержащего HTML-страницу. Сначала импортируем в src/index.js предоставляемую Electron функциональность `app` и `browserWindow`.

```
const { app, BrowserWindow } = require('electron');
```

Теперь можно определить для нашего приложения `browserWindow`, а также файл, который будет загружен. Добавьте в src/index.js следующее:

```
const { app, BrowserWindow } = require('electron');

// Чтобы не собирать мусор, объявляем window в виде переменной
```

```
let window;

// Указываем детали окна браузера
function createWindow() {
  window = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: true
    }
  });

  // Загружаем HTML-файл
  window.loadFile('index.html');

  // При закрытии окна сбрасываем объект
  window window.on('closed', () => {
    window = null;
  });
}

// Когда electron готов, создаем окно приложения
app.on('ready', createWindow);
```

Теперь мы готовы к запуску нашего десктопного приложения локально. Находясь в терминале, выполните из директории проекта следующее:

```
$ npm start
```

Эта команда выполнит `electron src/index.js`, запустив версию нашего приложения для среды разработки (рис. 18.1).

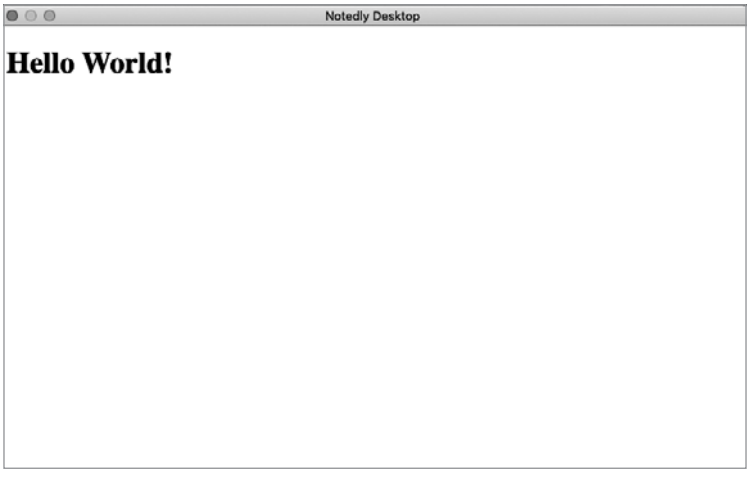

Рис. 18.1. Запуск приложения Hello World командой start

Детали окна приложения для macOS

В macOS окна приложения обрабатываются не так, как в Windows. Когда пользователь нажимает кнопку close window, окно закрывается, но само приложение продолжает работу. Если кликнуть по иконке приложения в панели dock macOS, окно приложения откроется снова. Electron позволяет нам реализовать такую функциональность. Добавьте в конец файла src/index.js следующее:

```
// Выходим при закрытии всех окон
app.on('window-all-closed', () => {
  // В macOS выходим, только если пользователь явно закрывает приложение
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

app.on('activate', () => {
  // В macOS повторно создаем окно при нажатии иконки в панели dock
  if (window === null) {
    createWindow();
  }
});
```

Добавив все это, вы увидите изменения при выходе и повторном входе в приложение с помощью команды `npm start`. Если теперь пользователь обратится к нашему приложению через macOS, то при закрытии окна он увидит ожидаемое поведение.

Инструменты разработчика

Поскольку в основе Electron лежит движок Chromium (используемый в Chrome, Microsoft Edge, Opera и многих других браузерах (ru.wikipedia.org/wiki/Браузеры_на_базе_Chromium)), он также дает доступ к инструментам разработчика Chromium. Это позволяет выполнять ту же отладку JavaScript, которую мы можем делать в среде браузера. Давайте настроим проверку нахождения приложения в режиме разработки. Если это так, будем автоматически открывать инструменты разработчика при его запуске.

Для выполнения этой проверки мы используем библиотеку `electron-util` (https://oreil.ly/JAf2Q). Она представляет собой небольшую коллекцию утилит, позволяющих легко проверять условия системы и в то же время упрощающих рутинный код для стандартных шаблонов Electron. Пока что мы будем использовать модуль `is`, который позволит проверить, находится ли приложение в режиме разработки.

Импортируйте модуль в начало файла src/index.js.

```
const { is } = require('electron-util');
```

Теперь можно добавить нижеприведенный код после `window.loadFile` (index.html), отвечающий за загрузку HTML-файла, чтобы открывать инструменты разработчика, если приложение находится в режиме разработки (рис. 18.2).

```
// Если приложение в режиме разработки, открываем браузерные инструменты
// разработчика
if (is.development) {
  window.веbContents.openDevTools();
}
```

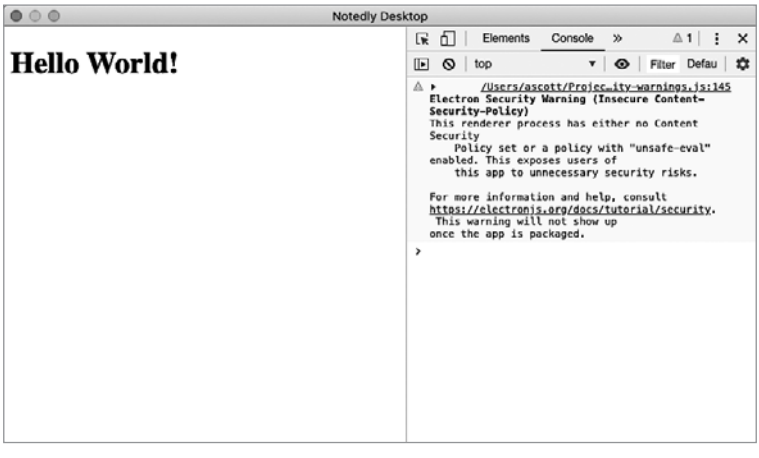

Рис. 18.2. Теперь при запуске приложения в процессе разработки вместе с ним будут запускаться инструменты разработчика

ПРЕДУПРЕЖДЕНИЕ БЕЗОПАСНОСТИ ELECTRON

Вы могли заметить, что сейчас наше приложение отображает предупреждение, связанное с небезопасной политикой защиты контента. Мы разберемся с этим в следующей главе.

Настроив удобный доступ к инструментам разработчика, мы готовы переходить к разработке клиентского приложения.

Electron API

Одно из преимуществ десктопной разработки в том, что с помощью Electron API мы получаем доступ к недоступным в среде браузера возможностям операционной системы, включая:

- уведомления;
- перетаскивание собственных файлов;

- режим Dark в macOS;
- пользовательские меню;
- надежные сочетания клавиш;
- системные диалоги;
- панель приложений;
- системную информацию.

Как вы понимаете, эти опции позволяют добавлять уникальные возможности и улучшать пользовательский опыт для десктопных клиентов. В нашем простом образце приложения мы их использовать не будем, но познакомиться с ними все же стоит. Документация Electron предоставляет подробные примеры всех Electron API. Кроме того, разработчики Electron (https://electronjs.org/docs) создали полнофункциональное приложение `electron-api-demos` (https://oreil.ly/Xo7NM), демонстрирующее многие из уникальных возможностей Electron API.

Итоги

В этой главе мы изучили основы использования Electron для построения десктопных приложений при помощи веб-технологий. Среда Electron дает нам как разработчикам возможность предоставить пользователям кроссплатформенные десктопные приложения без необходимости изучать премудрости разных языков программирования и операционных систем. Освоив простую настройку, рассмотренную в этой главе, и имея навык веб-разработки, мы полностью готовы к построению надежных десктопных приложений. В следующей главе мы рассмотрим, как внедрить существующее веб-приложение в оболочку Electron.

ГЛАВА 19

Интеграция веб-приложения в Electron

Я по привычке накапливаю вкладки в браузере, как ребенок, собирающий ракушки на пляже. Делаю я это не намеренно, но в конце дня их количество достигает нескольких десятков, распределенных по разным окнам браузера. Гордости это у меня не вызывает, но я чувствую, что такая привычка присуща многим. В итоге я использую десктопные версии некоторых из наиболее часто используемых веб-приложений. Зачастую эти приложения не имеют преимуществ в сравнении с их веб-версиями, но при этом к ним легче обращаться, их проще находить и на них удобно переключаться в течение дня.

В этой главе мы рассмотрим, как можно обернуть существующее веб-приложение в оболочку Electron. Прежде чем перейти к этому, потребуется сделать локальную копию наших образцов API и веб-приложений. Если вы не следовали всему процессу разработки на протяжении книги, то загляните в приложения А и Б, чтобы их запустить.

Интеграция веб-приложения

В предыдущей главе мы настроили приложение Electron на загрузку файла index.html. В качестве альтернативы можно загружать конкретный URL. В нашем случае мы начнем с загрузки URL нашего локально выполняемого приложения. Для начала убедитесь, что ваше веб-приложение и API запущены локально. Затем можно обновить файл src/index.js, начав с установки `nodeIntegration` в `BrowserWindow` как `false`. Это поможет избежать рисков безопасности при обращении локального узлового приложения к внешнему сайту.

```
веbPreferences: {
  nodeIntegration: false
},
```

Теперь замените строку `window.loadFile('index.html');` на следующую:

```
window.loadURL('http://localhost:1234');
```

ЗАПУСК ВЕБ-ПРИЛОЖЕНИЯ

Локальный экземпляр веб-приложения нужно будет запустить на порте 1234. Если вы выполняли все предыдущие инструкции из книги, то для запуска сервера разработки выполните из корня директории этого приложения команду **`npm start`**.

Это даст Electron команду выполнять загрузку URL, а не файла. Если теперь вы запустите приложение с помощью `npm start`, то увидите его загруженным в окне Electron с некоторыми замечаниями.

Предупреждения и ошибки

Инструменты разработчика Electron и наш терминал сейчас отображают большое количество предупреждений и ошибок. Давайте все их рассмотрим (рис. 19.1).

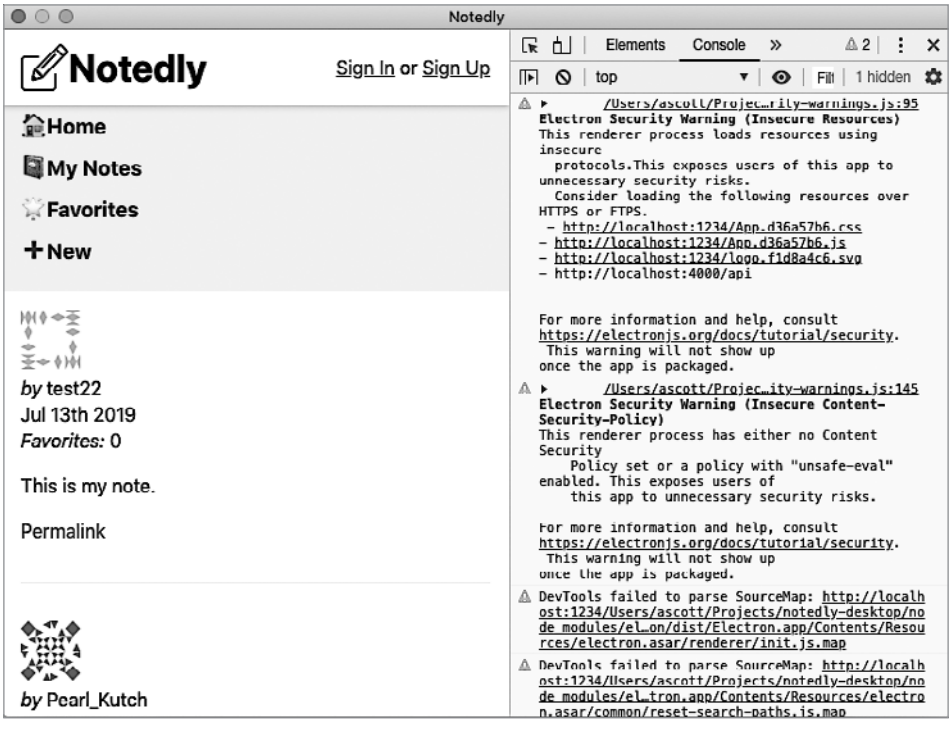

Рис. 19.1. Приложение работает, но отображает много ошибок и предупреждений

Во-первых, терминал отображает большое количество ошибок `SyntaxError: Unexpected Token`. Помимо этого, инструменты разработчика показывают несколько соответствующих предупреждений `DevTools failed to parse SourceMap`. Эти два вида ошибок относятся к способу, которым Parcel генерирует карты исходного кода, которые затем считывает Electron. К сожалению, при совмещении используемых нами технологий решить данную проблему невозможно. Поэтому наилучшим вариантом будет отключить карты исходного кода JavaScript. В инструментах разработчика окна приложения перейдите в раздел Settings и снимите галочку на Enable JavaScript source maps (рис. 19.2).

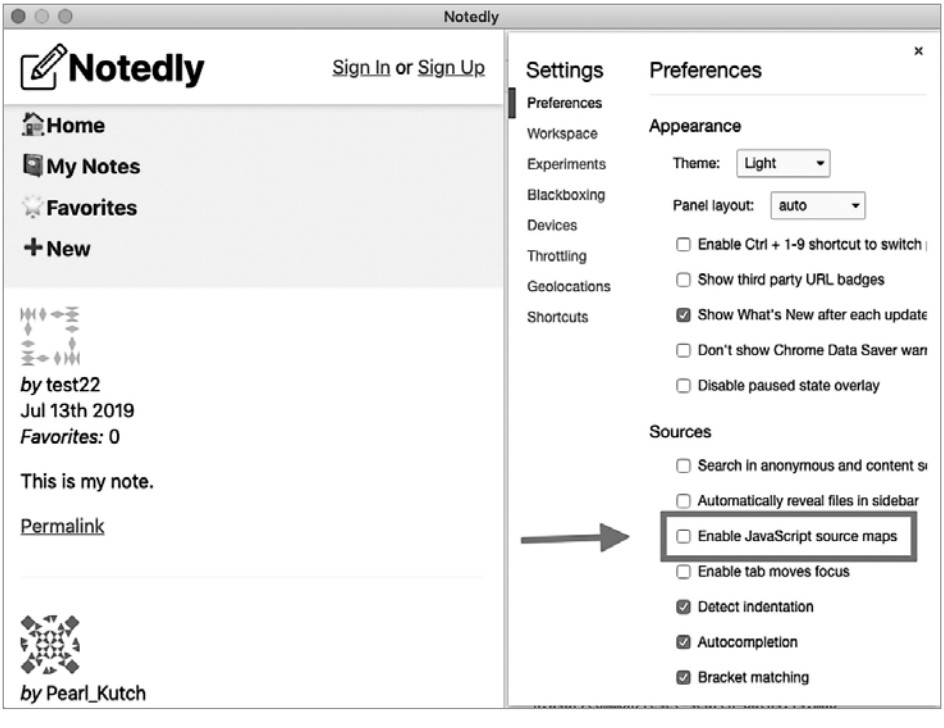

Рис. 19.2. Отключение карт исходного кода уменьшит число ошибок и предупреждений

Теперь, если вы выйдете из приложения и перезапустите его, то уже не увидите связанных с картами кода ошибок. Взамен мы получим несколько усложненный процесс отладки клиентского JS-кода в Electron, но, к счастью, все еще имеем доступ к этой возможности и нашему приложению в браузере.

Два последних предупреждения связаны с безопасностью Electron. Мы устраним их перед тем, как собирать приложение для производственной среды, но суть этих предупреждений лучше рассмотреть сейчас.

`Electron Security Warning (Insecure Resources)`

Это предупреждение сообщает, что мы загружаем веб-ресурсы через http-соединение. В производственной среде следует всегда загружать ресурсы через https в целях обеспечения конфиденциальности и безопасности. В процессе разработки загрузка локального хоста через http не является проблемой, так как мы будем ссылаться на размещенный веб-сайт, который использует в собранном приложении https.

`Electron Security Warning (Insecure Content-Security-Policy)`

Это предупреждение сообщает, что у нас еще не установлена политика безопасности контента (Content Security Policy, CSP). CSP позволяет указывать, с каких доменов нашему приложению разрешено загружать ресурсы, что существенно снижает риск атак межсайтового скриптинга (XSS). И, опять же, при локальной разработке это не является проблемой, но становится актуальным в производственной среде. Реализацией CSP мы займемся в этой главе позднее.

Разобравшись с ошибками, можно переходить к настройке файла конфигурации нашего приложения.

Конфигурация

В процессе локальной разработки нам нужна возможность запуска локальной версии веб-приложения, но при сборке этого приложения для стороннего использования нам нужно, чтобы оно ссылалось на публичный URL. Для этого можно настроить простой файл конфигурации.

Мы добавим в директорию ./src файл config.js, в котором будем хранить свойства приложения. Я включил файл config.example.js, который вы можете скопировать из терминала.

```
cp src/config.example.js src/config.js
```

Теперь можно прописать в нем свойства нашего приложения:

```
const config = {
  LOCAL_ВЕБ_URL: 'http://localhost:1234/',
  PRODUCTION_ВЕБ_URL: 'https://YOUR_DEPLOYED_ВЕБ_APP_URL',
  PRODUCTION_API_URL: 'https://YOUR_DEPLOYED_API_URL'
};

module.exports = config;
```

ПОЧЕМУ НЕ .ENV?

В предыдущих средах для управления их настройками мы использовали файлы .env. В данном случае мы задействуем файл конфигурации JavaScript из-за способа, которым приложения Electron распределяют свои зависимости.

Теперь в главном процессе приложения Electron мы можем использовать этот файл конфигурации, чтобы указать, какой URL хотим загружать для разработки или запуска в производство. В файле src/index.js сначала импортируйте файл config.js:

```
const config = require('./config');
```

Теперь можно обновить функциональность loadURL, чтобы загружать отдельный URL для каждой среды:

```
// Загружаем URL
if (is.development) {
  window.loadURL(config.LOCAL_ВЕБ_URL);
} else {
  window.loadURL(config.PRODUCTION_ВЕБ_URL);
}
```

Используя файл конфигурации, мы можем легко предоставить Electron настройки среды.

Политика защиты контента CSP

Как уже говорилось в этой главе, CSP позволяет указать домены, с которых приложению будет разрешено загружать ресурсы. Это помогает ограничить потенциальные XSS-атаки и атаки внедрения данных. В Electron можно указать настройки CSP для повышения степени безопасности приложения. Чтобы более подробно изучить использование CSP для Electron и веб-приложений, обратитесь к статье MDN (https://oreil.ly/VZS1H).

Electron предоставляет для CSP встроенный API, но библиотека `electron-util` предлагает более простой и понятный синтаксис. В начале файла src/index.js обновите инструкцию импорта `electron-util`, включив в нее `setContentSecurityPolicy`:

```
const { is, setContentSecurityPolicy } = require('electron-util');
```

Теперь можно установить CSP для продакшен-версии приложения:

```
// Устанавливаем CSP в производственном режиме
if (!is.development) {
  setContentSecurityPolicy(`
  default-src 'none';
  script-src 'self';
  img-src 'self' https://www.gravatar.com;
  style-src 'self' 'unsafe-inline';
  font-src 'self';
  connect-src 'self' ${config.PRODUCTION_API_URL};
  base-uri 'none';
```

```
  form-action 'none';
  frame-ancestors 'none';
`);
}
```

Написав CSP, мы можем выполнить проверку на ошибки, используя инструмент CSP Evaluator (https://oreil.ly/1xNK1). Если мы намеренно обращаемся к ресурсам по дополнительным URL-адресам, то можем добавить их в набор правил CSP.

В итоге файл src/index.js будет выглядеть так:

```
const { app, BrowserWindow } = require('electron');
const { is, setContentSecurityPolicy } = require('electron-util');
const config = require('./config');

// Чтобы не собирать мусор, объявляем window как переменную
let window;

// Указываем детали окна браузера
function createWindow() {
  window = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      nodeIntegration: false
    }
  });

// Загружаем URL
if (is.development) {
  window.loadURL(config.LOCAL_ВЕБ_URL);
} else {
  window.loadURL(config.PRODUCTION_ВЕБ_URL);
}

// Если находимся в режиме разработки, открываем инструменты разработчика
if (is.development) {
  window.webContents.openDevTools();
}

// Устанавливаем CSP в производственном режиме
if (!is.development) {
  setContentSecurityPolicy(`
  default-src 'none';
  script-src 'self';
  img-src 'self' https://www.gravatar.com;
  style-src 'self' 'unsafe-inline';
  font-src 'self';
  connect-src 'self' ${config.PRODUCTION_API_URL};
  base-uri 'none';
  form-action 'none';
  frame-ancestors 'none';
```

```
`);
}

// Когда окно закрыто, разыменовываем объект window
window.on('closed', () => {
  window = null;
  });
}

// Когда electron готов, создаем окно приложения
app.on('ready', createWindow);

// Выходим при закрытии всех окон
app.on('window-all-closed', () => {
  // В macOS выходим, только когда пользователь явно выходит из приложения
  if (process.platform !== 'darwin') {
    app.quit();
  }
});

app.on('activate', () => {
  // В macOS повторно создаем окно при клике на иконке в панели dock
  if (window === null) {
    createWindow();
  }
});
```

Теперь у нас есть рабочая реализация веб-приложения, выполняемого в оболочке Electron (рис. 19.3).

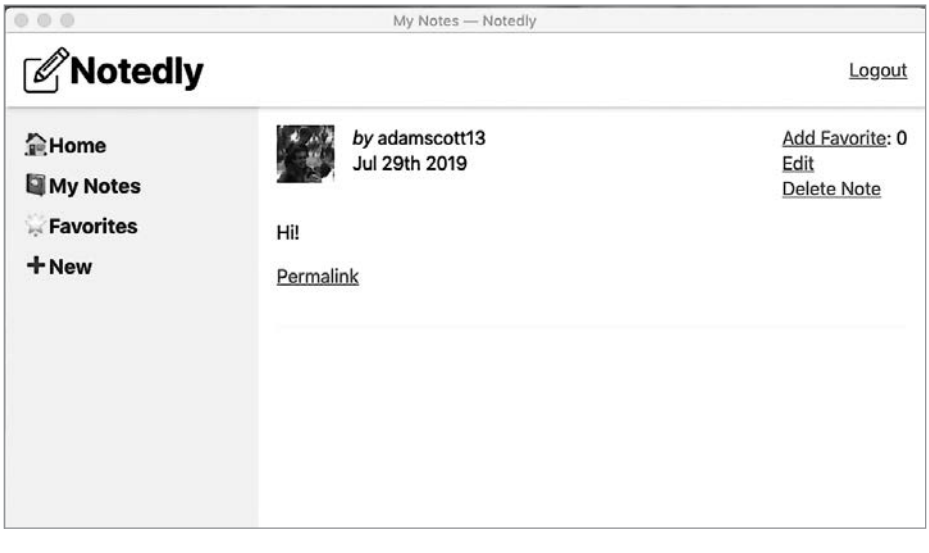

Рис. 19.3. Наше веб-приложение, выполняемое в оболочке Electron

Итоги

В этой главе мы интегрировали существующее веб-приложение в десктопное приложение Electron, что позволяет в кратчайшие сроки его выпустить. Стоит отметить, что в этом подходе есть и обратные стороны, поскольку он предлагает ограниченные десктопные преимущества и требует интернет-соединения для доступа ко всем функциям приложения. Для тех же, кто стремится поскорее выдвинуть свое приложение на рынок, эти недостатки могут быть оправданны. В следующей главе мы рассмотрим сборку и распространение приложения Electron.

ГЛАВА 20

Развертывание Electron

Когда я впервые вел курс по программированию, у меня возникла хорошая идея преподавать темы курса через текстовую приключенческую игру. Студенты должны были приходить в лабораторию, усаживаться за парты и проходить через серию забавных, на мой взгляд, подсказок и инструкций. Реакции на такой подход были неоднозначны, причем не из-за шуток, хотя, может быть, и из-за них тоже, а скорее потому, что учащимся ранее не приходилось взаимодействовать с «программой» таким способом. Они привыкли к GUI (графический интерфейс пользователя), и взаимодействие через текстовые подсказки для многих ощущалось как «неправильное».

Сейчас для запуска приложения нам нужно вводить команду в терминале, чтобы запустить процесс Electron. В этой главе мы рассмотрим, как можно собрать приложение для его распространения. Осуществим мы это с помощью популярной библиотеки Electron Builder (https://www.electron.build), которая поможет нам упаковать и предоставить приложение пользователям.

Electron Builder

Electron Builder — это библиотека, спроектированная для упрощения упаковки и дистрибуции приложений Electron и Proton Native (https://proton-native.js.org). Несмотря на то что для этих задач существуют и другие решения, Electron Builder упрощает ряд сложных моментов, связанных с дистрибуцией приложения, включая:

- подписание кода;
- мультиплатформенные получатели данных;
- автообновление;
- распространение.

Она предлагает отличный баланс гибкости и возможностей. Кроме того, хотя мы их использовать и не будем, в ней есть ряд шаблонов Electron Builder

для Webpack (https://oreil.ly/faYta), React (https://oreil.ly/qli_e), Vue (https://oreil.ly/9QY2W) и Vanilla JavaScript (https://oreil.ly/uJo7e).

ELECTRON BUILDER ИЛИ ELECTRON FORGE

Electron Forge (https://www.electronforge.io) — это еще одна популярная библиотека, предлагающая много схожих с Electron Builder возможностей. Главное преимущество Electron Forge в том, что в ее основе лежат официальные библиотеки Electron, в то время как Electron Builder является независимым инструментом сборки. Это означает, что пользователи получают преимущества параллельно с ростом экосистемы Electron. Недостаток же этой библиотеки в том, что она предусматривает гораздо более жесткую настройку приложения. Для задач этой книги Electron Builder предоставляет правильный баланс функций и возможностей обучения, но я все же рекомендую вам присмотреться и к Electron Forge.

Настройка Electron Builder

Вся настройка Electron Builder будет происходить в файле package.json нашего приложения. В нем можно увидеть, что `electron-builder` уже присутствует среди зависимостей разработки. В этот файл можно добавить ключ `build`, который будет содержать все инструкции Electron Builder для упаковки приложения. Для начала мы включим два поля:

`appId`

Это уникальный идентификатор нашего приложения, который в macOS называется `CFBundle Identifier` (https://oreil.ly/OOg1O), а в Windows — `AppUserModelID` (https://oreil.ly/mr9si). Для его создания стандартно используют обратный формат DNS. Например, если мы управляем компанией с доменом jseverywhere.io и соберем приложение под названием Notedly, то его ID будет `io.jseverywhere.notedly`.

`productName`

Это понятная человеку версия названия нашего продукта, поскольку поле `name` файла package.json требует использования имен, написанных через дефис или состоящих из одного слова.

В целом наша стартовая конфигурация будет выглядеть так:

```
"build": {
  "appId": "io.jseverywhere.notedly",
  "productName": "Notedly"
},
```

Electron Builder предоставляет множество вариантов настройки, некоторые из них мы рассмотрим в этой главе. Полный же их список вы можете найти в документации Electron Builder (https://oreil.ly/ESAx-).

Сборка для нашей текущей платформы

Настроив минимальную конфигурацию, мы можем создать первую сборку приложения. По умолчанию Electron Builder произведет ее для системы, в которой мы ведем разработку. Например, так как я пишу приложение на MacBook, по умолчанию сборка будет произведена для macOS.

Сначала давайте добавим в файл package.json два сценария, которые будут отвечать за сборку приложения. Первым будет сценарий `pack`, генерирующий директорию пакета без полной упаковки приложения. Это может пригодиться для тестирования. Второй — это сценарий `dist`, который будет упаковывать приложение в распространяемый формат, например в macOS DMG, установщик Windows или пакет DEB.

```
"scripts": {
// Добавляем сценарии pack и dist в существующий список сценариев npm
"pack": "electron-builder --dir",
"dist": "electron-builder"
}
```

Внеся это изменение, вы можете выполнить в терминале команду `npm run dist`, которая упакует приложение в директорию dist/ проекта. Перейдя в эту директорию, вы увидите, что Electron Builder упаковал приложение в дистрибутив для вашей операционной системы.

Иконки приложения

Вы могли заметить, что наше приложение использует стандартную иконку приложения Electron. Это годится при локальной разработке, но в производственной среде мы предпочтем использовать собственную символику продукта. Я включил в каталог /resources проекта ряд иконок приложения для macOS и Windows. Для генерации этих иконок из PNG-файла я использовал приложение iConvert Icons (https://iconverticons.com), которое доступно как для macOS, так и для Windows.

В каталоге /resources вы увидите следующие файлы:

- icon.icns, иконка приложения macOS;
- icon.ico, иконка приложения Windows;
- каталог icons с набором файлов .png разного размера, используемых в Linux.

При желании мы можем также включить фоновые изображения для macOS DMG, добавив иконки с именами background.png и background@2x.png для экранов Retina.

Теперь в файле package.json обновим объект `build`, указав имя директории с ресурсами для сборки:

```
"build": {
  "appId": "io.jseverywhere.notedly",
  "productName": "Notedly",
  "directories": {
    "buildResources": "resources"
  }
},
```

Теперь при сборке приложения Electron Builder упакует его с нашими пользовательскими иконками (рис. 20.1).

Рис. 20.1. Наша пользовательская иконка приложения на панели dock в macOS

Сборка для нескольких платформ

Пока что мы собираем приложение только для ОС, соответствующей платформе разработки. Одно из весомых преимуществ Electron как платформы — это то, что она позволяет использовать один и тот же код для нескольких платформ посредством обновления сценария `dist`. Для этого Electron Builder задействует бесплатный открытый сервис `electron-build-service` (https://oreil.ly/IEIfW). Мы будем использовать общедоступный экземпляр этого сервиса, но его также можно разместить и на собственном хостинге для организаций, заинтересованных в дополнительной безопасности и конфиденциальности.

Обновите сценарий `dist` файла package.json следующим образом:

```
"dist": "electron-builder -mwl"
```

Это приведет к сборке для платформ macOS, Windows и Linux. Теперь мы можем распространять наше приложение, выгружая его в качестве релиза на GitHub или в любое другое место дистрибуции вроде Amazon S3 или собственного веб-сервера.

Подписание кода

И macOS, и Windows предусматривают принцип подписания кода. Этот принцип повышает безопасность и доверие пользователей, поскольку помогает подтвер-

дить надежность приложения. Я не буду углубляться в детали этого процесса, так как он индивидуален для каждой ОС и требует определенных затрат от разработчиков. В документации Electron Builder есть исчерпывающий раздел (https://oreil.ly/g6wEz), посвященный подписанию кода для различных платформ.

Помимо этого, в документации к Electron (https://oreil.ly/Yb4JF) представлено несколько ресурсов и ссылок. Если вы собираете приложение для производственной среды, я рекомендую получше изучить варианты подписания кода для macOS и Windows.

Итоги

Мы рассмотрели лишь верхушку айсберга развертывания приложения Electron. В этой главе мы использовали для сборки библиотеку Electron Builder. После этого можно с легкостью выгружать и распространять приложения через любой веб-хостинг. Как только наши потребности возрастут, мы сможем использовать Electron Builder для интеграции сборок в конвейер непрерывной поставки продукта, автоматически отправлять релизы на GitHub, S3 или другие платформы дистрибуции, а также внедрять в наше приложение автоматические обновления. Если вы заинтересованы в дальнейшем изучении тем разработки на Electron и дистрибуции приложений, то освоение перечисленных действий будет очень кстати.

ГЛАВА 21

Мобильные приложения на React Native

Как-то раз, еще в конце 80-х, я прогуливался по магазинам с родителями и обратил внимание на портативный телевизор. Он представлял собой работающую на батарейках коробочку с антенной, небольшим динамиком и черно-белым экраном. Я был потрясен возможностью смотреть субботние утренние мультики на заднем дворе. Хоть у меня никогда такого устройства не было, одно только знание о существовании подобных вещей вызвало во мне ощущение, будто я живу в футуристическом мире. Тогда мне было невдомек, что, будучи взрослым, я буду носить в кармане устройство, которое не только даст мне возможность смотреть «Властелинов Вселенной», но и предоставит доступ к бескрайнему объему информации, позволит слушать музыку, играть в игры, делать заметки, фотографировать, вызывать такси, совершать покупки и выполнять множество других задач.

В 2007 году Стив Джобс представил миру iPhone, заявив, что «время от времени появляется революционный продукт, изменяющий все». Бесспорно, смартфоны существовали и до 2007 года, но до появления iPhone они не были в полном смысле слова «смарт», то есть умными. За прошедшие годы приложения для смартфонов уже прошли начальную стадию золотой лихорадки под девизом «все сойдет» и перешли на уровень, когда пользователи стали требовать от них качества и ожидать многого. Современные приложения преследуют высокие стандарты функциональности, удобства использования и дизайна. Задача также усложняется тем, что их разработка разделена между платформами Apple iOs и Android, каждая из которых использует разные языки программирования и инструменты.

Наверное, вы уже догадались (да и в названии книги это написано), что JavaScript дает нам возможность писать кроссплатформенные мобильные приложения. В этой главе мы познакомимся с предоставляющей эту возможность библиотекой React Native, а также с набором инструментов Expo. Помимо этого, мы клонируем код примера проекта, на основе которого будем вести разработку в течение нескольких ближайших глав.

Что мы создаем

На протяжении нескольких следующих глав мы будем разрабатывать мобильный клиент для нашего приложения социальных заметок Notedly. Целью будет использовать JS и веб-технологии для создания такого приложения, которое пользователи смогут устанавливать на мобильные устройства. Чтобы избежать излишнего повторения материала глав, посвященных веб-приложению, мы реализуем только подмножество функций. В частности, наше приложение будет:

- работать на iOS и Android;
- загружать из GraphQL API ленту заметок и заметки отдельного пользователя;
- использовать CSS и стилизованные компоненты для оформления стиля;
- выполнять стандартную и динамическую маршрутизацию.

Реализация этих возможностей позволит как следует представить ключевые принципы разработки мобильных приложений с помощью React Native. Прежде чем начать, давайте поближе взглянем на технологии, которые будем использовать.

Как мы будем это создавать

React Native — это основная технология, с помощью которой мы будем разрабатывать наше приложение. Она позволяет писать приложения на JavaScript, используя React, и отображать их для нативной мобильной платформы. Это означает, что для пользователей не будет ощутимой разницы между приложением React Native и написанным на языке платформы. Это главное ее преимущество перед другими популярными мобильными веб-фреймворками, которые традиционно обертывали веб-представление в оболочку приложения. React Native использовался для разработки своих приложений такими компаниями, как Facebook, Instagram, Bloomberg, Tesla, Skype, Walmart, Pinterest и многими другими.

Вторым ключевым элементом в создании нашего приложения будет Expo — набор инструментов и сервисов, упрощающих разработку с помощью ряда полезных возможностей, включающих локальный предпросмотр, сборку приложения и расширение основной библиотеки React Native. Прежде чем перейти к самой разработке, я советую вам сделать следующее.

1. Создать аккаунт на expo.io (https://expo.io/).
2. Установить инструменты командной строки Expo, набрав команду **`npm install expo-cli --global`** в терминале.

3. Войти в аккаунт Expo локально, введя **`expo login`** в терминале.
4. Установить приложение Expo Client для мобильных устройств. Ссылки на приложение для iOS и Android можно найти по адресу expo.io/tools.
5. Войти в аккаунт в приложении Expo Client.

И последнее. Для взаимодействия с данными из GraphQL API мы снова будем использовать Apollo Client (https://oreil.ly/xR62T), поскольку он представляет собой набор иструментов с открытым исходным кодом для работы с GraphQL.

Начало

Перед началом разработки вам потребуется сделать копию стартовых файлов проекта на компьютер. Исходный код проекта (https://github.com/javascripteverywhere/mobile) содержит все сценарии и ссылки на сторонние библиотеки, которые понадобятся нам в разработке. Чтобы скопировать код на локальную машину, откройте терминал, перейдите в директорию с проектами и выполните **`git clone`** для репозитория проекта. Если вы также следовали разработке в главах, посвященных API, веб- и десктопным приложениям, то у вас уже может быть создана директория notedly для упорядочивания кода проекта.

```
$ cd Projects
$ # Если у вас еще нет директории notedly, введите команду `mkdir notedly`
$ cd notedly
$ git clone git@github.com:javascripteverywhere/mobile.git
$ cd mobile
$ npm install
```

УСТАНОВКА СТОРОННИХ ЗАВИСИМОСТЕЙ

Сделав копию стартового кода проекта и выполнив `npm install` в директории, вы избавляетесь от необходимости повторно выполнять эту команду для каждой отдельной сторонней зависимости.

Структура кода:

/src

Директория, предназначенная для ведения разработки по ходу изучения книги.

/solutions

Директория, содержащая решения для каждой главы. Если у вас возникнут сложности, можете обратиться к ней за ответом.

/final

Директория для финальной версии рабочего проекта.

Остальные файлы и настройка проекта соответствуют стандартному выводу React Native-генератора `expo-cli`, который вы можете запустить из терминала командой **`expo init`**.

APP.JS?

В связи со спецификой работы цепочки сборки Expo файл App.js, расположенный в корневой директории, обычно выступает в качестве точки входа в приложение. Для стандартизации мобильного проекта с кодом из остальной части книги файл App.js используется только как ссылка на файл /src/Main.js.

Скопировав код на локальный компьютер и установив необходимые зависимости, начнем запуск приложения. Для этого введите в терминале следующее:

```
$ npm start
```

Эта команда откроет веб-приложение Expo Metro Bundler на локальном порте браузера. Отсюда вы можете запускать симулятор локального устройства нажатием на ссылки Run on.... Вы также можете запускать приложение на любом физическом устройстве с Expo Client, отсканировав QR-код (рис. 21.1).

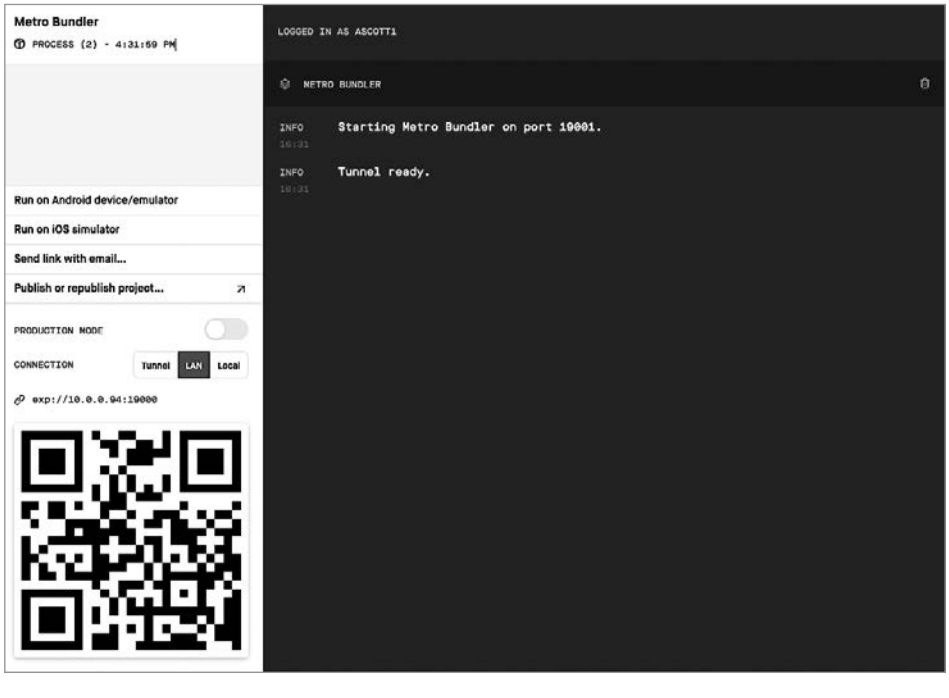

Рис. 21.1. Metro Bundler после запуска приложения

УСТАНОВКА СИМУЛЯТОРА УСТРОЙСТВ

Для запуска симулятора устройств iOS вам потребуется скачать и установить Xcode (https://oreil.ly/bgde4) (только для macOS). Для Android скачайте Android Studio (https://oreil.ly/bjqkn) и следуйте руководству Expo по настройке (https://oreil.ly/cUGsr). Сравнение симуляторов представлено на рис. 21.2. Но если вы только начинаете заниматься разработкой мобильных приложений, я рекомендую использовать собственное физическое устройство.

Рис. 21.2. Выполнение приложения на симуляторах устройств iOS и Android

Если вы авторизовались в Expo из терминала компьютера либо в Expo Client с мобильного устройства, то можете запустить приложение, просто нажав на вкладку Projects приложения Expo Client (рис. 21.3).

Скопировав код на локальную машину и имея возможность выполнять предпросмотр приложения в Expo Client, вы полностью готовы к началу разработки мобильного приложения.

Рис. 21.3. В Expo Client можно выполнять предпросмотр приложения на физическом устройстве

Итоги

Эта глава представила вам React Native и Expo. Мы клонировали код образца проекта, выполнили его локально на физическом устройстве или в симуляторе. React Native расширяет возможности веб- и JS-разработчиков, позволяя создавать полнофункциональные нативные мобильные приложения при помощи привычных инструментов и навыков. Expo упрощает набор инструментов и снижает порог входа в область нативной мобильной разработки. С помощью этих двух инструментов новички могут легко начать создавать мобильные приложения, а опытные веб-команды — быстро сформировать набор навыков для мобильной разработки. В следующей главе мы подробнее познакомимся с возможностями React Native, а также добавим в наше приложение стили и маршруты.

ГЛАВА 22

Оболочка мобильного приложения

Моя жена фотограф, а это значит, что большая часть ее жизни проходит за оформлением изображений в прямоугольной рамке. В фотографии есть много переменных — субъект, свет, угол, — но при этом пропорции снимка остаются постоянными. В условиях этого ограничения формируется удивительное разнообразие, определяющее то, как мы видим и запоминаем мир вокруг нас. Разработка мобильных приложений предоставляет аналогичные возможности. В рамках небольшого прямоугольного экрана мы можем создавать невероятно мощные приложения, предоставляя пользователям эффект глубокого погружения.

В данной главе мы начнем разработку оболочки нашего приложения. Для этого мы сначала получше познакомимся с ключевыми составляющими компонентов React Native. Затем мы рассмотрим применение стилей к приложению как с помощью встроенной поддержки в React Native, так и с помощью знакомой нам CSS-in-JS-библиотеки, Styled Components. Научившись применять стили, мы перейдем к интеграции маршрутизации. А в завершение изучим легкий способ дополнения интерфейса приложения иконками.

Из чего состоит React Native

Давайте рассмотрим основные составляющие элементы приложения React Native. Вы уже могли догадаться, что это приложение будет состоять из компонентов React, написанных на JSX. Но без DOM (объектной модели документа) HTML-страницы, что конкретно находится в этих компонентах? Можно начать с рассмотрения компонента Hello World в файле src/Main.js. Пока что я удалил из него стили.

```
import React from 'react';
import { Text, View } from 'react-native';

const Main = () => {
  return (
    <View>
```

```
      <Text>Hello world!</Text>
    </View>
  );
};

export default Main;
```

В этой разметке есть два примечательных JSX-тега: `<View>` и `<Text>`. Если вы уже имеете опыт веб-разработки, то тег `<View>` во многом служит той же цели, что и тег `<div>`. Он является контейнером для содержимого нашего приложения. Сами по себе они ничего не делают, но при этом содержат все наполнение приложения, могут вкладываться друг в друга и использоваться для применения стилей. Каждый компонент будет содержаться в теге `<View>`. В React Native вы можете использовать `<View>` везде, где обычно используете `<div>` или `<span>` в веб-разработке. Тег `<Text>` очевидно служит в качестве контейнера текста приложения. Однако в отличие от веба, для всего текста используется только он один.

Помимо этого, мы также можем добавлять в приложение изображения, для чего используется JSX-элемент `<Image>`. Давайте обновим файл src/Main.js, добавив в него изображение. Для этого мы импортируем из React Native компонент `Image` и используем тег `<Image>` с атрибутом `src` (рис. 22.1).

```
import React from 'react';
import { Text, View, Image } from 'react-native';

const Main = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Hello world!</Text>
      <Image source={require('../assets/images/hello-world.jpg')} />
    </View>
  );
};

export default Main;
```

Этот код отображает в представлении текст и изображение. Вы могли заметить, что наши JSX-теги `<View>` и `<Image>` являются передаваемыми свойствами, позволяющими нам контролировать конкретное поведение (в данном случае стиль представления и источник изображения). Передача свойств элементу позволяет добавлять ему различные дополнительные возможности. В документации API React Native (https://oreil.ly/3fACI) перечислены доступные для каждого элемента свойства.

Пока что наше приложение небогато возможностями, но в следующем разделе мы рассмотрим, как можно улучшить его внешний вид при помощи встроенной в React Native поддержки стилей и библиотеки Styled Components.

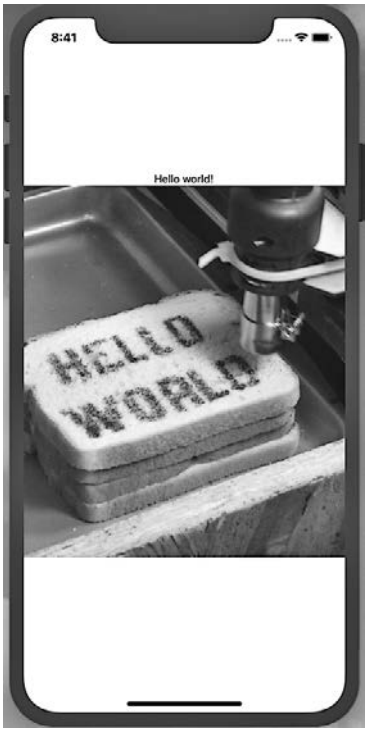

Рис. 22.1. Используя тег <Image>, мы можем добавлять в приложение изображения (фото сделано Винделлом Оскейем (Windell Oskay) (https://oreil.ly/IkW3F))

Style и Styled Components

Как разработчикам и дизайнерам нам нужна возможность стилизовать приложения, придавая им четкий вид и удобство для пользователей. Существует множество библиотек UI-компонентов вроде NativeBase (https://nativebase.io) или React Native Elements (https:// oreil.ly/-M8EE), которые предлагают широкий спектр предустановленных и часто настраиваемых компонентов. Все они достойны внимания, но для наших целей мы рассмотрим возможность составлять собственные стили и макеты приложения.

Как мы уже убедились, React Native предоставляет свойство `style`, позволяющее применять пользовательские стили к любому JSX-элементу приложения. Имена стилей и значения соответствуют их аналогам в CSS, за исключением того что имена пишутся в верблюжьем регистре, например `lineHeight` и `backgroundColor`. Давайте обновим файл /src/Main.js, включив некоторые стили для элемента `<Text>` (рис. 22.2).

```
const Main = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text style={{ color: '#0077cc', fontSize: 48, fontWeight: 'bold' }}>
         Hello world!
      </Text>
      <Image source={require('../assets/images/hello-world.jpg')} />
    </View>
  );
};
```

Вы можете справедливо заметить, что применение стилей на уровне элементов может быстро привести к неуправляемому хаосу. Поэтому можно использовать библиотеку React Native `StyleSheet`, чтобы организовать стили и использовать их повторно.

Сначала нам нужно добавить `StyleSheets` в список импортов (рис. 22.3).

```
import { Text, View, Image, StyleSheet } from 'react-native';
```

Рис. 22.2. С помощью стилей мы можем настраивать внешний вид элемента <Text>

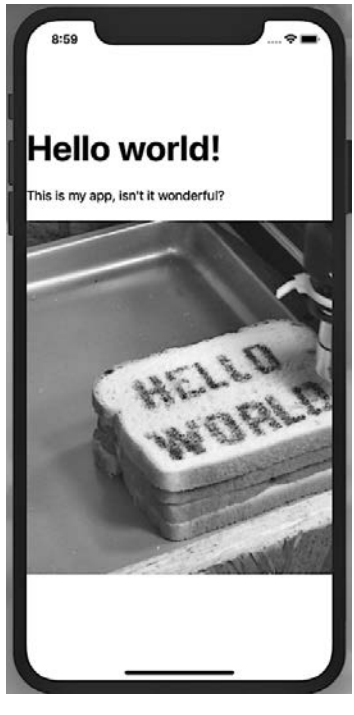

Рис. 22.3. Используя таблицы стилей, мы можем масштабировать стили приложения

Теперь можно абстрагировать стили.

```
const Main = () => {
  return (
    <View style={styles.container}>
      <Text style={styles.h1}>Hello world!</Text>
      <Text style={styles.paragraph}>This is my app</Text>
      <Image source={require('../assets/images/hello-world.jpg')} />
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center'
  },
  h1: {
    fontSize: 48,
    fontWeight: 'bold'
  },
  paragraph: {
    marginTop: 24,
    marginBottom: 24,
    fontSize: 18
  }
});
```

FLEXBOX

React Native для определения макетов стилей использует CSS-алгоритм flexbox. Мы не будем рассматривать его подробно, но React Native предлагает документацию (https://reactnative.dev/docs/flexbox), в которой четко поясняется, что это такое, а также приводятся варианты его использования для упорядочивания элементов на экране.

Styled Components

Несмотря на то что встроенные в React Native `StyleSheets` и свойства `style` могут предложить все необходимое по умолчанию, это далеко не единственные варианты стилизации приложений. Помимо них мы можем использовать такие популярные CSS-in-JS-решения, как Styled Components (https://www.styled-components.com) и Emotion (https://emotion.sh).

На мой взгляд, они предлагают более чистый синтаксис, ближе всего к CSS и ограничивают число необходимых переключений контекста между базами кода веб- и мобильных приложений. Применение этих веб-библиотек также дает возможность использовать одни и те же стили или компоненты на различных платформах.

Что касается конкретно наших целей, давайте посмотрим, как можно адаптировать предыдущий пример для использования библиотеки Styled Components. Сначала мы импортируем ее `native` версию в файл src/Main.js:

```
import styled from 'styled-components/native'
```

Отсюда мы можем перенести наши стили в синтаксис Styled Components. Если вы читали главу 13, то этот синтаксис должен показаться вам знакомым. Итоговый код файла src/Main.js будет следующим:

```
import React from 'react';
import { Text, View, Image } from 'react-native';
import styled from 'styled-components/native';

const StyledView = styled.View`
  flex: 1;
  justify-content: center;
`;

const H1 = styled.Text`
  font-size: 48px;
  font-weight: bold;
`;

const P = styled.Text`
  margin: 24px 0;
  font-size: 18px;
`;

const Main = () => {
  return (
    <StyledView>
      <H1>Hello world!</H1>
      <P>This is my app.</P>
      <Image source={require('../assets/images/hello-world.jpg')} />
    </StyledView>
  );
};

export default Main;
```

ИСПОЛЬЗОВАНИЕ ЗАГЛАВНЫХ БУКВ В STYLED COMPONENTS

В этой библиотеке все имена элементов должны писаться с заглавной буквы.

Теперь мы можем применять к приложению пользовательскую стилизацию, выбирая между встроенной системой стилей React Native и библиотекой Styled Components.

Маршрутизация

В интернете мы можем использовать в качестве привязок HTML-ссылки для связывания одного HTML-документа с другим, включая находящиеся на нашем собственном сайте. Для JS-приложений мы используем маршрутизацию, чтобы связать вместе шаблоны, отображаемые JavaScript. А что насчет нативных мобильных приложений? Для них мы будем перемещать пользователей между экранами. В этом разделе мы изучим два распространенных вида маршрутизации: перемещение по вкладкам и по стеку.

Маршрутизация по вкладкам с помощью React Navigation

Для реализации маршрутизации мы используем библиотеку React Navigation (https://reactnavigation.org/), рекомендуемую командами React Native и Expo. Самое главное, она существенно упрощает реализацию стандартных шаблонов маршрутизации с учетом особенностей платформы.

Для начала давайте создадим каталог screens в директории src. Затем в этом каталоге создадим три файла, каждый из которых будет содержать простейший компонент React.

Добавьте в src/screens/favorites.js следующее:

```
import React from 'react';
import { Text, View } from 'react-native';

const Favorites = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Favorites</Text>
    </View>
  );
};

export default Favorites;
```

В src/screens/feed.js добавьте следующее:

```
import React from 'react';
import { Text, View } from 'react-native';

const Feed = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Feed</Text>
    </View>
  );
};

export default Feed;
```

И наконец, добавьте следующее в src/screens/mynotes.js:

```
import React from 'react';
import { Text, View } from 'react-native';

const MyNotes = () => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>My Notes</Text>
    </View>
  );
};

export default MyNotes;
```

Затем мы можем создать новый файл src/screens/index.js, который будет использоваться в качестве корня маршрутизации нашего приложения. Начнем мы с импорта начальных зависимостей `react` и `react-navigation`.

```
import React from 'react';
import { createAppContainer } from 'react-navigation';
import { createBottomTabNavigator } from 'react-navigation-tabs';

// Импортируем компоненты экрана
import Feed from './feed';
import Favorites from './favorites';
import MyNotes from './mynotes';
```

Импортировав эти зависимости, мы можем создать навигатор по вкладкам между этими тремя экранами при помощи `createBottomTabNavigator`, чтобы определить, какие экраны компонента React должны появляться при перемещении.

```
const TabNavigator = createBottomTabNavigator({
  FeedScreen: {
    screen: Feed,
    navigationOptions: {
      tabBarLabel: 'Feed',
    }
  },
  MyNoteScreen: {
    screen: MyNotes,
    navigationOptions: {
      tabBarLabel: 'My Notes',
    }
  },
  FavoriteScreen: {
    screen: Favorites,
    navigationOptions: {
      tabBarLabel: 'Favorites',
    }
```

```
  }
});

// Создаем контейнер приложения
export default createAppContainer(TabNavigator);
```

В завершение обновляем файл src/Main.js, чтобы он только импортировал маршрутизатор. Теперь в упрощенном виде он должен выглядеть так:

```
import React from 'react';
import Screens from './screens';

const Main = () => {
  return <Screens />;
};

export default Main;
```

Убедитесь, что приложение запущено, набрав команду **`npm start`** в терминале. Теперь вы должны увидеть меню навигации по вкладкам в нижней части дисплея, при нажатии на которые будете перенаправляться на соответствующий экран (рис. 22.4).

Рис. 22.4. Теперь мы можем перемещаться между экранами с помощью навигации по вкладкам

Навигация по стеку

Второй вид маршрутизации — это навигация по стеку, при которой экраны образно составляются в стек один поверх другого, давая пользователям перемещаться по этому стеку вглубь и обратно. В качестве примера можно взять приложения новостей, где пользователь просматривает ленту. В этом случае он может нажать на заголовок одной из предлагаемых в ленте статей, перейдя вглубь стека для просмотра ее содержимого. После этого он может нажать «назад», вернувшись в ленту, или кликнуть на заголовке другой статьи, еще дальше углубившись в стек.

Мы хотим, чтобы в нашем приложении пользователи могли перемещаться из ленты заметок к самим заметкам и обратно. Давайте рассмотрим, как можно реализовать навигацию по стеку для каждого имеющегося у нас экрана.

Сначала давайте создадим компонент `NoteScreen`, который будет содержать второй экран стека. Создайте файл src/screensnote.js с минимальным компонентом React Native.

```
import React from 'react';
import { Text, View } from 'react-native';

const NoteScreen = () => {
  return (
    <View style={{ padding: 10 }}>
      <Text>This is a note!</Text>
    </View>
  );
};

export default NoteScreen;
```

Затем мы внесем изменения в маршрутизатор, чтобы активировать для компонента `NoteScreen` навигацию по стеку. Для этого мы импортируем `createStackNavigator` из `react-navigation-stack`, а также наш новый компонент note.js. Обновите импорты в src/screens/index.js следующим образом:

```
import React from 'react';
import { Text, View, ScrollView, Button } from 'react-native';
import { createAppContainer } from 'react-navigation';
import { createBottomTabNavigator } from 'react-navigation-tabs';
// Добавляем импорт для createStackNavigator
import { createStackNavigator } from 'react-navigation-stack';

// Импортируем компоненты экрана, включая note.js
import Feed from './feed';
import Favorites from './favorites';
import MyNotes from './mynotes';
import NoteScreen from './note';
```

Импортировав библиотеки и файлы, мы можем реализовать возможность навигации по стеку. В файле маршрутизатора необходимо сообщить React Navigation, какие экраны должны помещаться в стек. Нам нужно, чтобы из каждого закрепленного за вкладками маршрута пользователь мог переходить на экран `Note`. Определите эти стеки следующим образом:

```
const FeedStack = createStackNavigator({
  Feed: Feed,
  Note: NoteScreen
});

const MyStack = createStackNavigator({
  MyNotes: MyNotes,
  Note: NoteScreen
});

const FavStack = createStackNavigator({
  Favorites: Favorites,
  Note: NoteScreen
});
```

Теперь можно обновить `TabNavigator`, чтобы он ссылался на стек, а не на отдельный экран. Для этого обновите свойство `screen` в каждом объекте `TabNavigator`.

```
const TabNavigator = createBottomTabNavigator({
  FeedScreen: {
    screen: FeedStack,
    navigationOptions: {
      tabBarLabel: 'Feed'
    }
  },
  MyNoteScreen: {
    screen: MyStack,
    navigationOptions: {
      tabBarLabel: 'My Notes'
    }
  },
  FavoriteScreen: {
    screen: FavStack,
    navigationOptions: {
      tabBarLabel: 'Favorites'
    }
  }
});
```

В целом файл src/screens/index.js теперь должен выглядеть так:

```
import React from 'react';
import { Text, View, ScrollView, Button } from 'react-native';
import { createAppContainer } from 'react-navigation';
import { createBottomTabNavigator } from 'react-navigation-tabs';
import { createStackNavigator } from 'react-navigation-stack';

// Импортируем компоненты экрана
```

```
import Feed from './feed';
import Favorites from './favorites';
import MyNotes from './mynotes';
import NoteScreen from './note';

// Стек навигации
const FeedStack = createStackNavigator({
  Feed: Feed,
  Note: NoteScreen
});

const MyStack = createStackNavigator({
  MyNotes: MyNotes,
  Note: NoteScreen
});

const FavStack = createStackNavigator({
  Favorites: Favorites,
  Note: NoteScreen
});

// Вкладки навигации
const TabNavigator = createBottomTabNavigator({
  FeedScreen: {
    screen: FeedStack,
    navigationOptions: {
      tabBarLabel: 'Feed'
    }
  },
  MyNoteScreen: {
    screen: MyStack,
    navigationOptions: {
      tabBarLabel: 'My Notes'
    }
  },
  FavoriteScreen: {
    screen: FavStack,
    navigationOptions: {
      tabBarLabel: 'Favorites'
    }
  }
});

// Создаем контейнер приложения
export default createAppContainer(TabNavigator);
```

Если теперь мы откроем приложение в симуляторе или приложении Expo на нашем устройстве, то не увидим никакой заметной разницы. Это связано с тем, что нам еще нужно добавить ссылку на навигацию по стеку в компонент src/screens/feed.js. Давайте это сделаем.

Для этого сначала включите зависимость `Button` из React Native:

```
import { Text, View, Button } from 'react-native';
```

Теперь мы можем добавить кнопку, которая при нажатии будет выполнять переход к содержимому нашего компонента note.js. Мы передадим компонент `props`, который будет содержать информацию по навигации, и добавим `<Button>`, который будет включать свойства `title` и `onPress`.

```
const Feed = props => {
  return (
    <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
      <Text>Note Feed</Text>
      <Button
        title="Keep reading"
        onPress={() => props.navigation.navigate('Note')}
      />
    </View>
  );
};
```

После этого мы должны получить возможность перемещаться между экранами. Кликните по кнопке на экране Feed, чтобы перейти на экран Note, а затем кликните по стрелке для возвращения (рис. 22.5).

Рис. 22.5. Клик по кнопке будет вести пользователя на новый экран, а клик по стрелке будет возвращать его обратно

Добавление заголовков экранам

При добавлении навигации по стеку в верхнюю часть приложения автоматически добавляется строка заголовков. Мы можем стилизовать или даже удалить эту строку. Пока что давайте добавим заголовок каждому экрану в верхней части стека. Для этого мы установим `navigationOptions` вне самого компонента. В src/screens/feed.js:

```
import React from 'react';
import { Text, View, Button } from 'react-native';

const Feed = props => {
// Код компонента
};

Feed.navigationOptions = {
  title: 'Feed'
};

export default Feed;
```

Мы можем повторить этот процесс и для других компонентов экрана.

В src/screens/favorites.js:

```
Favorites.navigationOptions = {
  title: 'Favorites'
};
```

В src/screens/mynotes.js:

```
MyNotes.navigationOptions = {
  title: 'My Notes'
};
```

Теперь каждый экран будет содержать заголовок в верхней панели навигации (рис. 22.6).

Рис. 22.6. При установке заголовка в navigationOptions он будет добавлен в верхнюю панель навигации

Иконки

Сейчас функциональность навигации готова, но ей недостает визуального компонента, который бы улучшил пользовательский опыт. К счастью, Expo делает процесс добавления иконок очень простым. Мы можем найти все доступные для использования иконки по ссылке expo.github.io/vector-icons. При этом

ряд их наборов включен в этот инструмент изначально. К ним относятся Ant Design, Ionicons, Font Awesome, Entypo, Foundation, Material Icons и Material Community Icons. Таким образом, Expo предоставляет нам огромное оригинальное разнообразие.

Давайте добавим несколько иконок в нашу навигацию по вкладкам. Сначала нужно импортировать желаемые наборы иконок. В нашем случае мы используем Material Community Icons, добавив в файл src/screens/index.js следующее:

```
import { MaterialCommunityIcons } from '@expo/vector-icons';
```

Теперь везде, где мы захотим использовать в компоненте иконку, мы можем добавить ее как JSX, включая установку таких свойств, как `size` и `color`.

```
<MaterialCommunityIcons name="star" size={24} color={'blue'} />
```

Мы будем добавлять иконки в навигацию по вкладкам. React Navigation включает свойства `tabBarIcon`, позволяющие устанавливать иконку. Мы можем передать его в виде функции, что позволит нам установить `tintColor`, чтобы активные иконки вкладок отличались цветом от неактивных:

```
const TabNavigator = createBottomTabNavigator({
  FeedScreen: {
    screen: FeedStack,
    navigationOptions: {
      tabBarLabel: 'Feed',
      tabBarIcon: ({ tintColor }) => (
        <MaterialCommunityIcons name="home" size={24} color={tintColor} />
      )
    }
  },
  MyNoteScreen: {
    screen: MyStack,
    navigationOptions: {
      tabBarLabel: 'My Notes',
      tabBarIcon: ({ tintColor }) => (
        <MaterialCommunityIcons name="notebook" size={24} color={tintColor} />
      )
    }
  },
  FavoriteScreen: {
    screen: FavStack,
    navigationOptions: {
      tabBarLabel: 'Favorites',
      tabBarIcon: ({ tintColor }) => (
        <MaterialCommunityIcons name="star" size={24} color={tintColor} />
      )
    }
  }
});
```

Теперь навигация по вкладкам будет представлена с иконками (рис. 22.7).

Рис. 22.7. Теперь в навигации по приложению есть иконки

Итоги

В этой главе мы рассмотрели разработку простых компонентов приложения React Native. Теперь вы сможете создавать компоненты, добавлять к ним стили и перемещаться между ними. Надеюсь, что эта простая настройка позволит вам ощутить весь невероятный потенциал React Native. Имея минимальный набор новых технологий, вы уже можете создавать основы впечатляющих и профессиональных мобильных приложений. В следующей главе мы используем GraphQL для добавления в приложение данных из нашего API.

ГЛАВА 23

GraphQL и React Native

В музее Энди Уорхола (Andy Warhol) в Питтсбурге, штат Пенсильвания, есть инсталляция под названием «Серебряные облака». Она представляет собой небольшую комнату с примерно дюжиной прямоугольных шаров из фольги, заполненных смесью гелия и обычного воздуха. В итоге эти шары остаются в подвешенном состоянии дольше шаров, заполненных атмосферным воздухом, но при этом не взлетают к потолку, как их аналоги с гелием. Смотрители музея ходят по комнате и игриво подбрасывают эти шары, чтобы удержать их в воздухе.

Сейчас наше приложение во многом напоминает комнату с «облаками». Можно довольствоваться нажатием иконок и переходом по оболочке приложения, но в итоге оно все равно представляет собой пустую комнату (не в обиду мистеру Уорхолу). В этой главе мы начнем заполнять его, сперва освоив отображение контента при помощи представлений списков React Native. Затем мы используем Apollo Client (https://www.apollographql.com/docs/react) для подключения к API с данными. Подключившись, мы напишем GraphQL-запросы, которые будут отображать эти данные на экране приложения.

ЛОКАЛЬНЫЙ ЗАПУСК API

Для разработки мобильного приложения нам потребуется доступ к локальному экземпляру нашего API. Если вы следовали всем указаниям книги, то у вас на машине уже должен быть рабочий Notedly API с базой данных. Если нет, то я добавил в приложение A инструкции о том, как сделать копию рабочего API вместе с набором образцов данных. Если у вас уже есть API, но вы бы хотели добавить дополнительные данные, выполните **`npm run seed`** из корня директории проекта API.

Создание списка и прокручиваемого содержимого

Списки повсюду. В повседневной жизни мы составляем списки дел, покупок и гостей. В приложениях списки являются одним из наиболее распространен-

ных шаблонов UI: списки постов в социальных сетях, статей, песен, фильмов и т. д. Заметьте, что сейчас я тоже составил список. Поэтому вполне естественно, что React Native делает создание прокручиваемых списков контента очень простым.

В этой библиотеке есть два вида списков: `FlatList` и `SectionList`. Первый полезен для организации большого количества элементов в один прокручиваемый список. При этом React Native фоном выполняет ряд полезных действий, например отрисовывает только те элементы, которые изначально доступны для просмотра, повышая тем самым производительность. `SectionList` во многом похож на `FlatList`, за исключением того, что он позволяет группам элементов списка иметь заголовок. Например, контакты в списке контактов зачастую группируются по алфавиту под буквенно-цифровым заголовком.

В нашем случае мы будем использовать `FlatList`, чтобы отображать список заметок, который пользователь сможет прокручивать и нажимать на просмотр для чтения полной версии заметки.

Для реализации этого мы создадим новый компонент `NoteFeed`, который сможем использовать для отображения этого списка. Пока что мы будем использовать фиктивные данные, но вскоре подключимся к нашему API.

Как только что отмечалось, начнем мы с создания компонента src/components/NoteFeed.js. В него мы импортируем зависимости и добавим массив временных данных.

```
import React from 'react';
import { FlatList, View, Text } from 'react-native';
import styled from 'styled-components/native';

// Фиктивные данные
const notes = [
  {  id:  0,  content:  'Giant Steps' },
  {  id:  1,  content:  'Tomorrow Is The Question' },
  {  id:  2,  content:  'Tonight At Noon' },
  {  id:  3,  content:  'Out To Lunch' },
  {  id:  4,  content:  'Green Street' },
  {  id:  5,  content:  'In A Silent Way' },
  {  id:  6,  content:  'Lanquidity' },
  {  id:  7,  content:  'Nuff Said' },
  {  id:  8,  content:  'Nova' },
  {  id:  9,  content:  'The Awakening' }
];

const NoteFeed = () => {
// Здесь будет код компонента
};

export default NoteFeed;
```

Теперь можно написать код компонента, который будет содержать `FlatList`:

```
const NoteFeed = props => {
  return (
    <View>
      <FlatList
        data={notes}
        keyExtractor={({ id }) => id.toString()}
        renderItem={({ item }) => <Text>{item.content}</Text>}
      />
    </View>
  );
};
```

В этом коде видно, что `FlatList` получает три свойства, которые упрощают процесс перебора данных:

`data`

Это свойство указывает на массив данных, который будет содержать список.

`keyExtractor`

У каждого элемента списка должно быть уникальное значение `key`. Мы используем `keyExtractor`, чтобы задействовать в качестве `key` уникальное значение `id`.

`renderItem`

Это свойство определяет, что нужно отобразить в списке. Пока что мы передаем отдельный `item` из массива `notes` и отображаем его как `Text`.

Мы можем просмотреть наш список, обновив компонент src/screens/feed.js для отображения ленты:

```
import React from 'react';

// Импортируем NoteFeed
import NoteFeed from '../components/NoteFeed';

const Feed = props => {
  return <NoteFeed />;
};

Feed.navigationOptions = {
  title: 'Feed'
};

export default Feed;
```

Давайте вернемся к файлу src/components/NoteFeed.js и обновим `renderItem`, добавив интервал между элементами списка при помощи стилизованного компонента:

```
// Определение компонента в стиле FeedView
const FeedView = styled.View`
  height: 100;
  overflow: hidden;
  margin-bottom: 10px;
`;

const NoteFeed = props => {
  return (
    <View>
      <FlatList
        data={notes}
        keyExtractor={({ id }) => id.toString()}
        renderItem={({ item }) => (
          <FeedView>
            <Text>{item.content}</Text>
          </FeedView>
        )}
      />
    </View>
  );
};
```

Если вы выполните предпросмотр приложения, то увидите прокручиваемый список данных. Наконец, мы можем добавить разделитель между элементами списка. Вместо добавления нижней границы через CSS, React Native позволяет нам передать в наш `FlatList` свойство `ItemSeparatorComponent`, предоставив тем самым возможность использовать в качестве разделителя компонент любого типа. При этом мы также избегаем расположения разделителя в нежелательных местах, например после последнего элемента списка. В нашем случае мы добавим простую границу, созданную как стилизованный компонент `View`:

```
// Определение компонента в стиле FeedView
const FeedView = styled.View`
  height: 100;
  overflow: hidden;
  margin-bottom: 10px;
`;

// Добавляем компонент в стиле Separator (разделитель)
const Separator = styled.View`
  height: 1;
  width: 100%;
  background-color: #ced0ce;
```

```
`;

const NoteFeed = props => {
  return (
    <View>
      <FlatList
        data={notes}
        keyExtractor={({ id }) => id.toString()}
        ItemSeparatorComponent={() => <Separator />}
        renderItem={({ item }) => (
          <FeedView>
            <Text>{item.content}</Text>
          </FeedView>
        )}
      />
    </View>
  );
};
```

Вместо отрисовки и стилизации содержимого нашей заметки напрямую во `FlatList` давайте изолируем его в собственный компонент. Для этого введем новый тип представления под названием `ScrollView`. Его функциональность в точности такая, как вы могли догадаться: вместо того чтобы вписываться в размеры страницы, он выводит содержимое за ее границы, позволяя пользователям делать прокрутку.

Давайте создадим новый компонент src/components/Note.js:

```
import React from 'react';
import { Text, ScrollView } from 'react-native';
import styled from 'styled-components/native';

const NoteView = styled.ScrollView`
  padding: 10px;
`;

const Note = props => {
  return (
    <NoteView>
      <Text>{props.note.content}</Text>
    </NoteView>
  );
};

export default Note;
```

Наконец, чтобы использовать новый компонент `Note`, мы импортируем его и поместим в `FeedView` файла src/components/NoteFeed.js. Итоговый код компонента будет следующим (рис. 23.1):

```
import React from 'react';
import { FlatList, View, Text } from 'react-native';
import styled from 'styled-components/native';

import Note from './Note';

// Фиктивные данные
const notes = [
  { id: 0, content: 'Giant Steps' },
  { id: 1, content: 'Tomorrow Is The Question' },
  { id: 2, content: 'Tonight At Noon' },
  { id: 3, content: 'Out To Lunch' },
  { id: 4, content: 'Green Street' },
  { id: 5, content: 'In A Silent Way' },
  { id: 6, content: 'Lanquidity' },
  { id: 7, content: 'Nuff Said' },
  { id: 8, content: 'Nova' },
  { id: 9, content: 'The Awakening' }
];

// Определение компонента в стиле FeedView
const FeedView = styled.View`
  height: 100;
  overflow: hidden;
  margin-bottom: 10px;
`;

const Separator = styled.View`
  height: 1;
  width: 100%;
  background-color: #ced0ce;
`;

const NoteFeed = props => {
  return (
    <View>
      <FlatList
        data={notes}
        keyExtractor={({ id }) => id.toString()}
        ItemSeparatorComponent={() => <Separator />}
        renderItem={({ item }) => (
          <FeedView>
            <Note note={item} />
          </FeedView>
        )}
      />
    </View>
  );
};

export default NoteFeed;
```

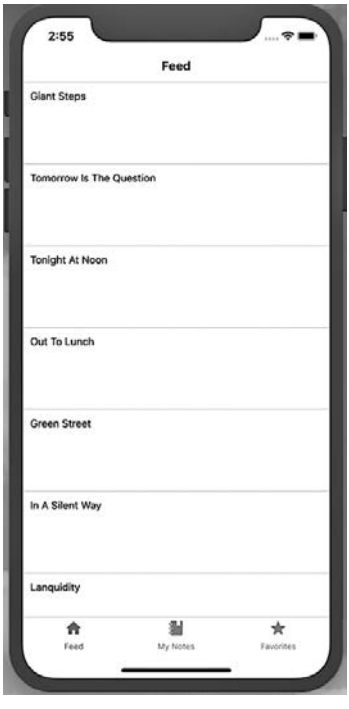

Рис. 23.1. При помощи FlatList мы можем отображать список данных

Таким образом мы создали простой `FlatList`. Теперь давайте сделаем возможным переход из элемента списка к отдельному маршруту.

Маршрутизация списка

В мобильных приложениях зачастую реализуется шаблон, по которому нажатие на элемент списка выводит дополнительную информацию или расширенную функциональность. Вы можете вспомнить из предыдущей главы, что наш экран ленты в стеке навигации располагается над экраном заметок. В React Native мы можем использовать в качестве обертки `TouchableOpacity`, чтобы представление отвечало на прикосновение пользователя к экрану. Это означает, что мы можем обернуть содержимое `FeedView` в `TouchableOpacity` и перенаправить пользователя при нажатии так же, как мы делали это ранее с помощью кнопки. Давайте продолжим, обновив компонент src/components/NoteFeed.js для выполнения именно этого действия.

Для начала нужно обновить импорт `react-native` в src/components/NoteFeed.js, включив в него `TouchableOpacity`:

```
import { FlatList, View, TouchableOpacity } from 'react-native';
```

Затем мы обновим компонент для использования `TouchableOpacity`:

```
const NoteFeed = props => {
  return (
    <View>
      <FlatList
        data={notes}
        keyExtractor={({ id }) => id.toString()}
        ItemSeparatorComponent={() => <Separator />}
        renderItem={({ item }) => (
          <TouchableOpacity
            onPress={() =>
              props.navigation.navigate('Note', {
                id: item.id
              })
            }
          >
            <FeedView>
              <Note note={item} />
            </FeedView>
          </TouchableOpacity>
        )}
      />
    </View>
  );
};
```

Нам также потребуется изменить компонент экрана feed.js для передачи свойств навигации в ленту. В src/screens/feed.js:

```
const Feed = props => {
  return <NoteFeed navigation={props.navigation} />;
};
```

Теперь мы можем легко перемещаться на наш общий экран заметок. Давайте настроим этот экран, чтобы он показывал ID заметки. Вы могли заметить, что в навигации по компоненту `Note Feed` мы передаем свойство ID. В файле screens/note.js можно прочесть значение этого свойства:

```
import React from 'react';
import { Text, View } from 'react-native';

const NoteScreen = props => {
  const id = props.navigation.getParam('id');
  return (
    <View style={{ padding: 10 }}>
      <Text>This is note {id}</Text>
    </View>
  );
};

export default NoteScreen;
```

Теперь мы можем переходить из представления списка к странице деталей. Закончив с этим, можно переходить к интеграции в приложение данных из нашего API.

GraphQL с Apollo Client

Пока что мы готовы читать и отображать данные в приложении. Далее мы будем обращаться к GraphQL API, который разработали в первой части книги. Для удобства мы будем использовать Apollo Client — ту же клиентскую библиотеку GraphQL, которую использовали в веб-разделе книги. Эта библиотека предлагает ряд полезных функций для упрощения работы с GraphQL в JS-приложениях UI. Клиентские возможности Apollo включают получение данных от удаленного API, локальное кэширование, обработку ошибок, управление локальным состоянием и многое другое.

Прежде всего нам понадобится настроить файл конфигурации. Мы будем хранить переменные среды в файле config.js. В React Native есть несколько способов управления переменными среды и конфигурации, но я считаю такой подход к конфигурированию наиболее простым и эффективным. Для начала я добавил файл config-example.js, который вы можете скопировать и отредактировать значениями из нашего приложения. В терминале из корня директории проекта выполните следующую команду:

```
$ cp config.example.js config.js
```

Здесь мы можем обновить любые переменные `dev` или `prod` среды. В нашем случае это будет только значение производственного API_URI.

```
// Устанавливаем переменные среды
const ENV = {
  dev: {
    API_URI: `http://${localhost}:4000/api`
  },
  prod: {
    // Обновляем значение API_URI на адрес публично развернутого API
    API API_URI: 'https://your-api-uri/api'
  }
};
```

Теперь мы сможем обращаться к этим двум значениям на основе среды Expo при помощи функции `getEnvVars`. Мы не будем углубляться в остальную часть файла конфигурации, но если вас заинтересует дальнейшее изучение настройки, вы найдете в нем подробные комментарии.

Закончив с этим, мы можем перейти к подключению нашего клиента к API. Мы установим Apollo в src/Main.js при помощи библиотеки Apollo Client. Если вы следовали инструкциям веб-раздела книги, то это будет вам знакомо.

```
import React from 'react';
import Screens from './screens';
// Импортируем библиотеки Apollo
import { ApolloClient, ApolloProvider, InMemoryCache } from '@apollo/client';
// Импортируем конфигурацию среды
import getEnvVars from '../config';
const { API_URI } = getEnvVars();

// Настраиваем API URI и кэш
const uri = API_URI;
const cache = new InMemoryCache();

// Настраиваем Apollo Client
const client = new ApolloClient({
  uri,
  cache
});

const Main = () => {
  // Оборачиваем приложение в компонент высшего порядка ApolloProvider
  return (
    <ApolloProvider client={client}>
      <Screens />
    </ApolloProvider>
  );
};

export default Main;
```

Все это не вызовет видимых изменений в приложении, но теперь мы подключены к нашему API. Далее мы рассмотрим, как запрашивать из этого API данные.

Написание GraphQL-запросов

Подключившись к API, можно переходить к запросу данных. Начнем мы с запроса всех заметок из базы данных для их отображения в списке `NoteFeed`. Затем запросим отдельные заметки для их отображения в представлении деталей `Note`.

ЗАПРОС ЗАМЕТОК

Для простоты и уменьшения повторения вместо постраничного запроса `noteFeed` мы будем использовать массовый запрос заметок из API.

Написание компонента `Query` происходит в точности так же, как и в веб-приложении React. Мы импортируем в файл src/screens/feed.js библиотеки `useQuery` и GraphQL Language (`gql`) следующим образом:

```
// Импортируем зависимости React Native и Apollo
import { Text } from 'react-native';
import { useQuery, gql } from '@apollo/client';
```

Теперь составляем запрос:

```
const GET_NOTES = gql`
  query notes {
    notes {
      id
      createdAt
      content
      favoriteCount
      author {
        username
        id
        avatar
      }
    }
  }
`;
```

В финале обновляем компонент для вызова запроса:

```
const Feed = props => {
  const { loading, error, data } = useQuery(GET_NOTES);

  // Если данные загружаются, наше приложение будет показывать индикатор
  // загрузки
  if (loading) return <Text>Loading</Text>;
  // Если при получении данных произошел сбой, отображаем сообщение об ошибке
  if (error) return <Text>Error loading notes</Text>;
  // Если запрос выполнен успешно и содержит заметки, возвращаем их в ленту
  return <NoteFeed notes={data.notes} navigation={props.navigation} />;
};
```

В целом файл src/screens/feed.js теперь выглядит так:

```
import React from 'react';
import { Text } from 'react-native';
// Импортируем библиотеки Apollo
import { useQuery, gql } from '@apollo/client';

import NoteFeed from '../components/NoteFeed';
import Loading from '../components/Loading';

// Составляем запрос
const GET_NOTES = gql`
  query notes {
    notes {
      id
      createdAt
      content
      favoriteCount
      author {
        username
```

```
        id
        avatar
      }
    }
  }
`;

const Feed = props => {
  const { loading, error, data } = useQuery(GET_NOTES);

  // Если данные загружаются, приложение будет показывать индикатор загрузки
  if (loading) return <Text>Loading</Text>;
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <Text>Error loading notes</Text>;
  // Если запрос выполнен успешно и содержит заметки, возвращаем их в ленту
  return <NoteFeed notes={data.notes} navigation={props.navigation} />;
};

Feed.navigationOptions = {
  title: 'Feed'
};

export default Feed;
```

Написав запрос, мы можем обновить компонент src/components/NoteFeed.js для использования данных, переданных через `props`:

```
const NoteFeed = props => {
  return (
    <View>
      <FlatList
        data={props.notes}
        keyExtractor={({ id }) => id.toString()}
        ItemSeparatorComponent={() => <Separator />}
        renderItem={({ item }) => (
          <TouchableOpacity
            onPress={() =>
              props.navigation.navigate('Note', {
                id: item.id
              })
            }
          >
            <FeedView>
              <Note note={item} />
            </FeedView>
          </TouchableOpacity>
        )}
      />
    </View>
    );
  };
```

С этими изменениями при запущенном Expo мы увидим отображение в списке данных из нашего API (рис. 23.2).

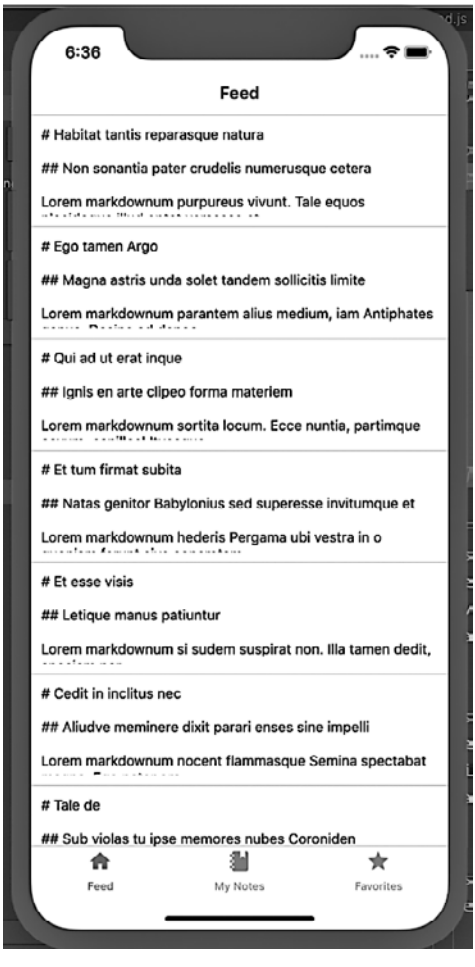

Рис. 23.2. Отображение данных API в ленте

Сейчас при нажатии на просмотр в списке по-прежнему будет отображаться обычная страница заметки. Давайте исправим это, создав в файле src/screens/note.js запрос `note`.

```
import React from 'react';
import { Text } from 'react-native';
import { useQuery, gql } from '@apollo/client';
```

```
import Note from '../components/Note';

// Запрос note, принимающий переменную ID
const GET_NOTE = gql`
  query note($id: ID!) {
    note(id: $id) {
      id
      createdAt
      content
      favoriteCount
      author {
        username
        id
        avatar
      }
    }
  }
`;

const NoteScreen = props => {
  const id = props.navigation.getParam('id');
  const { loading, error, data } = useQuery(GET_NOTE, { variables: { id } });

  if (loading) return <Text>Loading</Text>;
  // В случае сбоя выдаем пользователю сообщение об ошибке
  if (error) return <Text>Error! Note not found</Text>;
  // В случае успеха передаем данные в компонент note
  return <Note note={data.note} />;
};

export default NoteScreen;
```

В конце давайте обновим компонент src/components/Note.js для отображения содержимого заметки. Для этого мы добавим две зависимости, `react-markdown-renderer` и `date-fns`, чтобы считывать Markdown и даты из нашего API понятным для пользователя образом:

```
import React from 'react';
import { Text, ScrollView } from 'react-native';
import styled from 'styled-components/native';
import Markdown from 'react-native-markdown-renderer';
import { format } from 'date-fns';

const NoteView = styled.ScrollView`
  padding: 10px;
`;

const Note = ({ note }) => {
  return (
    <NoteView>
```

```
        <Text>
          Note by {note.author.username} / Published{' '}
          {format(new Date(note.createdAt), 'MMM do yyyy')}
        </Text>
        <Markdown>{note.content}</Markdown>
      </NoteView>
    );
};

export default Note;
```

Внеся эти изменения, мы увидим в ленте приложения список заметок. При выборе просмотра заметки мы увидим ее полное содержание с возможностью прокрутки (рис. 23.3).

Рис. 23.3. Написав GraphQL-запросы, мы можем перемещаться между экранами для предварительного или полного просмотра заметок

Добавление индикатора загрузки

Сейчас загрузка данных приложением сопровождается мигающим словом Loading. Оно понятно сообщает о происходящем, но зачастую такая подача раздражает пользователей. React Native предоставляет нам встроенную возможность `ActivityIndicator`, которая отображает соответствующий ОС спиннер загрузки. Давайте напишем простой компонент, который затем используем в качестве индикатора.

Создайте файл src/components/Loading.js и напишите простой компонент, отображающий индикатор активности в центре экрана:

```
import React from 'react';
import { View, ActivityIndicator } from 'react-native';
import styled from 'styled-components/native';

const LoadingWrap = styled.View`
  flex: 1;
  justify-content: center;
  align-items: center;
`;

const Loading = () => {
  return (
    <LoadingWrap>
      <ActivityIndicator size="large" />
    </LoadingWrap>
  );
};

export default Loading;
```

Теперь можно заменить текст Loading в компонентах GraphQL-запроса. Сначала импортируйте компонент Loading в src/screens/feed.js и src/screens/note.js:

```
import Loading from '../components/Loading';
```

Затем в этих же файлах обновите состояние загрузки Apollo:

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

Теперь при загрузке данных из API наше приложение будет отображать вращающийся индикатор активности (рис. 23.4).

Рис. 23.4. При помощи ActivityIndicator мы можем добавить подходящий для ОС спиннер загрузки

Итоги

В этой главе мы сначала рассмотрели интеграцию представлений списков в приложение React Native, использовав распространенные шаблоны UI. Затем настроили Apollo Client и интегрировали в приложение данные из нашего API. Теперь у нас есть все необходимое для построения самых распространенных типов приложений наподобие новостных лент или для интеграции ленты блога с сайта. В следующей главе мы добавим аутентификацию и отобразим пользовательские запросы.

ГЛАВА 24

Аутентификация в мобильном приложении

Если вам доводилось гостить у родственника, отдыхать в съемных апартаментах или арендовать меблированное жилье, то вы знаете, каково это — находиться в окружении чужих вещей. В такой обстановке сложно почувствовать себя комфортно, так как приходится следить за порядком и стараться ничего не нарушить. Когда я оказываюсь в подобных ситуациях, независимо от степени дружелюбия хозяина понимание, что все вокруг чужое, держит меня в постоянном напряжении. Что здесь можно сказать? Мне весьма некомфортно, если я не могу даже просто поставить стакан на стол без подставки.

Аналогичное чувство дискомфорта может возникнуть и у наших пользователей, если приложение не предоставит им возможность читать или настраивать собственные данные. Их заметки будут просто смешиваться в общую кучу со всеми остальными, не давая почувствовать себя их хозяином. В этой главе мы добавим в приложение аутентификацию, введем сохранение данных токена при помощи Expo-хранилища `SecureStore`, создадим в React Native текстовые формы и выполним GraphQL-мутации аутентификации.

Поток аутентификации

Давайте начнем с создания потока аутентификации. При первом обращении пользователя к приложению мы будем показывать ему экран входа в систему. После того как он авторизуется, мы будем сохранять токен на его устройстве, позволяя в дальнейшем миновать этот экран. Мы также добавим экран настроек, где пользователь сможет нажимать кнопку для выхода из приложения и удалять токен со своего устройства.

Для реализации всего этого мы поэтапно добавим несколько новых экранов:

authloading.js

Промежуточный экран, с которым пользователи взаимодействовать не будут. При открытии приложения мы будем использовать его для проверки наличия токена и перенаправлять пользователя либо на экран авторизации, либо в само приложение.

signin.js

Это экран, где пользователь может войти в свой аккаунт. После успешной попытки входа мы будем сохранять токен на его устройстве.

settings.js

На экране настроек пользователь сможет нажать кнопку и выйти из приложения, после чего будет перенаправлен на экран авторизации.

ИСПОЛЬЗОВАНИЕ СУЩЕСТВУЮЩЕЙ УЧЕТНОЙ ЗАПИСИ

Возможность создания учетной записи через приложение мы добавим позже в этой главе. Если вы еще этого не сделали, то будет полезным создать учетную запись либо напрямую в вашем экземпляре API в GraphQL Playground, либо через интерфейс веб-приложения.

Для хранения токенов и работы с ними мы будем использовать Expo-библиотеку SecureStore (https://oreil.ly/nvqEO). На мой взгляд, это хороший способ шифрования и хранения данных на локальном устройстве. Для iOS она использует сервисы связок ключей (https://oreil.ly/iCu8R), а для Android — объект Shared Preferences операционной системы, шифруя данные при помощи Keystore (https://oreil.ly/gIXsp). Все это происходит фоном, позволяя нам просто сохранять и извлекать данные.

Для начала создадим экран авторизации. Пока что он будет состоять из компонента `Button`, который при нажатии будет сохранять токен. Давайте создадим новый компонент экрана src/screens/signin.js и импортируем в него зависимости:

```
import React from 'react';
import { View, Button, Text } from 'react-native';
import * as SecureStore from 'expo-secure-store';

const SignIn = props => {
  return (
    <View>
      <Button title="Sign in!" />
    </View>
  );
}
```

```
SignIn.navigationOptions = {
  title: 'Sign In'
};

export default SignIn;
```

Теперь создадим компонент загрузки аутентификации в src/screens/authloading.js, который пока будет просто отображать индикатор загрузки:

```
import React, { useEffect } from 'react';
import * as SecureStore from 'expo-secure-store';

import Loading from '../components/Loading';

const AuthLoading = props => {
  return <Loading />;
};

export default AuthLoading;
```

И наконец, создадим экран настроек в src/screens/settings.js:

```
import React from 'react';
import { View, Button } from 'react-native';
import * as SecureStore from 'expo-secure-store';

const Settings = props => {
  return (
    <View>
      <Button title="Sign Out" />
    </View>
  );
};

Settings.navigationOptions = {
  title: 'Settings'
};

export default Settings;
```

Написав эти компоненты, мы обновим нашу маршрутизацию для обработки аутентифицированных и неаутентифицированных пользователей. Добавьте новые экраны в список инструкций импорта файла в src/screens/index.js:

```
import AuthLoading from './authloading';
import SignIn from './signin';
import Settings from './settings';
```

Нам также потребуется обновить зависимость `react-navigation`, включив в нее `createSwitchNavigator`, позволяющий отображать по одному экрану за раз и переключаться между ними. При навигации пользователя `SwitchNavigator`

сбрасывает маршруты до состояния по умолчанию, не предлагая возможности перейти обратно.

```
import { createAppContainer, createSwitchNavigator } from 'react-navigation';
```

Для экранов аутентификации и настроек можно создать `StackNavigator`. Это позволит при необходимости добавлять экраны поднавигации в будущем.

```
const AuthStack = createStackNavigator({
  SignIn: SignIn
});

const SettingsStack = createStackNavigator({
  Settings: Settings
});
```

Далее мы добавим экран настроек в нижнюю часть — `TabNavigator`. Остальная часть настроек навигации по вкладкам останется прежней:

```
const TabNavigator = createBottomTabNavigator({
  FeedScreen: {
    // ...
  },
  MyNoteScreen: {
    // ...
  },
  FavoriteScreen: {
    // ...
  },
  Settings: {
    screen: Settings,
    navigationOptions: {
      tabBarLabel: 'Settings',
      tabBarIcon: ({ tintColor }) => (
        <MaterialCommunityIcons name="settings" size={24} color={tintColor} />
      )
    }
  }
});
```

Теперь мы можем создать `SwitchNavigator`, определив экраны для переключения и установив `AuthLoading` в качестве экрана по умолчанию. После этого мы заменим имеющийся оператор `export` на экспортирующий `SwitchNavigator`:

```
const SwitchNavigator = createSwitchNavigator(
  {
    AuthLoading: AuthLoading,
    Auth: AuthStack,
    App: TabNavigator
  },
  {
```

```
    initialRouteName: 'AuthLoading'
  }
);

export default createAppContainer(SwitchNavigator);
```

В итоге файл src/screens/index.js будет выглядеть так:

```
import React from 'react';
import { Text, View, ScrollView, Button } from 'react-native';
import { createAppContainer, createSwitchNavigator } from 'react-navigation';
import { createBottomTabNavigator } from 'react-navigation-tabs';
import { createStackNavigator } from 'react-navigation-stack';
import { MaterialCommunityIcons } from '@expo/vector-icons';

import Feed from './feed';
import Favorites from './favorites';
import MyNotes from './mynotes';
import Note from './note';
import SignIn from './signin';
import AuthLoading from './authloading';
import Settings from './settings';

const AuthStack = createStackNavigator({
  SignIn: SignIn,
});

const FeedStack = createStackNavigator({
  Feed: Feed,
  Note: Note
});

const MyStack = createStackNavigator({
  MyNotes: MyNotes,
  Note: Note
});

const FavStack = createStackNavigator({
  Favorites: Favorites,
  Note: Note
});

const SettingsStack = createStackNavigator({
  Settings: Settings
});

const TabNavigator = createBottomTabNavigator({
  FeedScreen: {
    screen: FeedStack,
    navigationOptions: {
      tabBarLabel: 'Feed',
      tabBarIcon: ({ tintColor }) => (
        <MaterialCommunityIcons name="home" size={24} color={tintColor} />
```

```
        )
      }
    },
    MyNoteScreen: {
      screen: MyStack,
      navigationOptions: {
        tabBarLabel: 'My Notes',
        tabBarIcon: ({ tintColor }) => (
          <MaterialCommunityIcons name="notebook" size={24} color={tintColor} />
        )
      }
    },
    FavoriteScreen: {
      screen: FavStack,
      navigationOptions: {
        tabBarLabel: 'Favorites',
        tabBarIcon: ({ tintColor }) => (
          <MaterialCommunityIcons name="star" size={24} color={tintColor} />
        )
      }
    },
    Settings: {
      screen: SettingsStack,
      navigationOptions: {
        tabBarLabel: 'Settings',
        tabBarIcon: ({ tintColor }) => (
          <MaterialCommunityIcons name="settings" size={24} color={tintColor} />
        )
      }
    }
  });

  const SwitchNavigator = createSwitchNavigator(
    {
      AuthLoading: AuthLoading,
      Auth: AuthStack,
      App: TabNavigator
    },
    {
      initialRouteName: 'AuthLoading'
    }
  );

  export default createAppContainer(SwitchNavigator);
```

Сейчас при предпросмотре приложения мы увидим только спиннер загрузки, так как маршрут `AuthLoading` является стартовым экраном. Давайте изменим это, чтобы экран загрузки проверял существование значения `token` в `SecureStore` приложения. При обнаружении токена мы будем направлять пользователя на главный экран, а при его отсутствии — на экран авторизации. Давайте обновим файл src/screens/authloading.js для выполнения этой проверки:

```
import React, { useEffect } from 'react';
import * as SecureStore from 'expo-secure-store';

import Loading from '../components/Loading';

const AuthLoadingScreen = props => {
  const checkLoginState = async () => {
    // Извлекаем значение токена
    const userToken = await SecureStore.getItemAsync('token');
    // Если токен найден, переходим на экран приложения
    // В противном случае переходим на экран авторизации
    props.navigation.navigate(userToken ? 'App' : 'Auth');
  };

  // Вызываем checkLoginState, как только компонент установится
  useEffect(() => {
    checkLoginState();
  });

  return <Loading />;
};

export default AuthLoadingScreen;
```

После этих изменений во время загрузки приложения при отсутствии токена мы должны перенаправляться на экран авторизации. Пока что давайте обновим этот экран, чтобы он хранил общий токен и переходил в приложение при нажатии кнопки пользователем (рис. 24.1):

```
import React from 'react';
import { View, Button, Text } from 'react-native';
import * as SecureStore from 'expo-secure-store';

const SignIn = props => {
  // Сохраняем токен со значением ключа `token`
  // После сохранения токена переходим на главный экран приложения
  const storeToken = () => {
    SecureStore.setItemAsync('token', 'abc').then(
       props.navigation.navigate('App')
    );
  };

  return (
    <View>
      <Button title="Sign in!" onPress={storeToken} />
    </View>
  );
};

SignIn.navigationOptions = {
  title: 'Sign In'
};

export default SignIn;
```

Рис. 24.1. Нажатие на кнопку будет сохранять токен и направлять пользователя в приложение

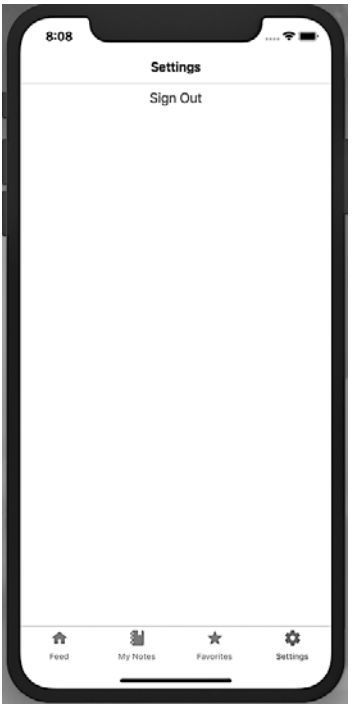

Рис. 24.2. Нажатие на кнопку приведет к удалению токена с устройства и возвращению пользователя на экран авторизации

Теперь токен сохраняется в `SecureStore` при нажатии на кнопку. Закончив реализацию правильной работы авторизации, давайте предоставим пользователям возможность выходить из приложения. Для этого мы добавим на экран настроек кнопку, которая при нажатии будет удалять токен из `SecureStore` (рис. 24.2).

В src/screens/settings.js:

```
import React from 'react';
import { View, Button } from 'react-native';
import * as SecureStore from 'expo-secure-store';

const Settings = props => {
  // Удаляем токен и переходим на экран авторизации
  const signOut = () => {
    SecureStore.deleteItemAsync('token').then(
      props.navigation.navigate('Auth')
    );
  };
```

```
    return (
      <View>
        <Button title="Sign Out" onPress={signOut} />
      </View>
    );
  };

  Settings.navigationOptions = {
    title: 'Settings'
  };

  export default Settings;
```

Теперь у нас есть все необходимое для создания потока аутентификации приложения.

НЕ ЗАБУДЬТЕ ВЫЙТИ ИЗ ПРИЛОЖЕНИЯ

Если вы еще этого не сделали, нажмите в локальном экземпляре приложения кнопку Sign Out. В последующих разделах мы добавим соответствующую функциональность авторизации.

Создание формы авторизации

Несмотря на то что теперь мы можем нажать на кнопку и сохранить токен на устройстве пользователя, мы еще не предоставили ему возможность авторизовываться, вводя собственную информацию. Давайте исправим это, начав с создания формы, в которой пользователь сможет вводить имейл и пароль. Для этого мы создадим компонент в src/components/UserForm.js с формой, используя React-Native-компонент `TextInput`.

```
  import React, { useState } from 'react';
  import { View, Text, TextInput, Button, TouchableOpacity } from 'react-
native';
  import styled from 'styled-components/native';

  const UserForm = props => {
    return (
      <View>
        <Text>Email</Text>
        <TextInput />
        <Text>Password</Text>
        <TextInput />
        <Button title="Log In" />
      </View>
    );
  }

  export default UserForm;
```

Теперь мы можем отобразить эту форму на экране авторизации. Для этого обновите файл src/screens/signin.js для импорта и использования компонента следующим образом:

```
import React from 'react';
import { View, Button, Text } from 'react-native';
import * as SecureStore from 'expo-secure-store';

import UserForm from '../components/UserForm';

const SignIn = props => {
  const storeToken = () => {
    SecureStore.setItemAsync('token', 'abc').then(
       props.navigation.navigate('App')
    );
  };
  return (
    <View>
      <UserForm />
    </View>
  );
}

export default SignIn;
```

Теперь на экране авторизации мы увидим отображение простой формы, но ей недостает стилизации и функциональности. Можно продолжить реализацию этой формы в файле src/components/UserForm.js. Мы используем React-хук `useState` для считывания и установки значений элементов формы.

```
const UserForm = props => {
  // Состояние элемента формы
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();

  return (
    <View>
      <Text>Email</Text>
      <TextInput onChangeText={text => setEmail(text)} value={email} />
      <Text>Password</Text>
      <TextInput onChangeText={text => setPassword(text)} value={password} />
      <Button title="Log In" />
    </View>
  );
}
```

Теперь можно добавить элементам формы несколько дополнительных свойств, чтобы предоставить пользователям ожидаемую функциональность при работе с имейлами или паролями. Полная документация по API `TextInput` доступна в руководстве React Native (https://oreil.ly/yvgyU). Мы будем также вызывать функцию при нажатии кнопки, хотя функциональность при этом будет ограничена.

```
const UserForm = props => {
  // Состояние элемента формы
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();

  const handleSubmit = () => {
    // Эта функция вызывается при нажатии пользователем кнопки формы
  };

  return (
    <View>
      <Text>Email</Text>
      <TextInput
        onChangeText={text => setEmail(text)}
        value={email}
        textContentType="emailAddress"
        autoCompleteType="email"
        autoFocus={true}
        autoCapitalize="none"
      />
      <Text>Password</Text>
      <TextInput
        onChangeText={text => setPassword(text)}
        value={password}
        textContentType="password"
        secureTextEntry={true}
      />
      <Button title="Log In" onPress={handleSubmit} />
    </View>
  );
}
```

Теперь у нашей формы есть все необходимые компоненты, но ее оформление оставляет желать лучшего. Давайте используем библиотеку Styled Components, чтобы придать нашей форме более подходящий вид:

```
import React, { useState } from 'react';
import { View, Text, TextInput, Button, TouchableOpacity } from 'react-
native';
import styled from 'styled-components/native';

const FormView = styled.View`
  padding: 10px;
`;

const StyledInput = styled.TextInput`
  border: 1px solid gray;
  font-size: 18px;
  padding: 8px;
  margin-bottom: 24px;
`;

const FormLabel = styled.Text`
```

```
  font-size: 18px;
  font-weight: bold;
`;

const UserForm = props => {
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();

  const handleSubmit = () => {
    // Эта функция вызывается, когда пользователь нажимает кнопку формы
  };

  return (
    <FormView>
      <FormLabel>Email</FormLabel>
      <StyledInput
        onChangeText={text => setEmail(text)}
        value={email}
        textContentType="emailAddress"
        autoCompleteType="email"
        autoFocus={true}
        autoCapitalize="none"
      />
      <FormLabel>Password</FormLabel>
      <StyledInput
        onChangeText={text => setPassword(text)}
        value={password}
        textContentType="password"
        secureTextEntry={true}
      />
      <Button title="Log In" onPress={handleSubmit} />
    </FormView>
  );
};

export default UserForm;
```

В итоге наш компонент Button будет ограничен предустановленными опциями стиля, за исключением значения свойства `color`. Чтобы создать компонент кнопки с настраиваемым стилем, мы можем использовать React Native-обертку `TouchableOpacity` (рис. 24.3).

```
const FormButton = styled.TouchableOpacity`
  background: #0077cc;
  width: 100%;
  padding: 8px;
`;

const ButtonText = styled.Text`
  text-align: center;
  color: #fff;
  font-weight: bold;
```

```
  font-size: 18px;
`;

const UserForm = props => {
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();

  const handleSubmit = () => {
    // Эта функция вызывается, когда пользователь нажимает кнопку формы
  };

  return (
    <FormView>
      <FormLabel>Email</FormLabel>
      <StyledInput
        onChangeText={text => setEmail(text)}
        value={email}
        textContentType="emailAddress"
        autoCompleteType="email"
        autoFocus={true}
        autoCapitalize="none"
      />
      <FormLabel>Password</FormLabel>
      <StyledInput
        onChangeText={text => setPassword(text)}
        value={password}
        textContentType="password"
        secureTextEntry={true}
      />
      <FormButton onPress={handleSubmit}>
        <ButtonText>Submit</ButtonText>
      </FormButton>
    </FormView>
  );
};
```

Таким образом, мы реализовали форму авторизации и применили к ней настраиваемые стили. Теперь давайте перейдем к реализации ее функциональности.

Аутентификация с помощью GraphQL-мутаций

Вы можете вспомнить нашу разработку потока аутентификации в главах, посвященных API и веб-приложениям, но, прежде чем двигаться дальше, давайте все же освежим знания. Мы будем отправлять в наш API GraphQL-мутацию, включающую имейл пользователя и пароль. Если указанный имейл будет найден в базе данных и пароль окажется верным, API отправит в ответ JWT. После этого мы сможем сохранить этот токен на устройстве пользователя, как уже делали, и отправлять его с каждым GraphQL-запросом. Это позволит нам идентифицировать пользователя при каждом запросе к API, не требуя повторного ввода пароля.

Рис. 24.3. Форма авторизации с настраиваемыми стилями

Создав форму, мы можем написать в файле src/screens/signin.js GraphQL-мутацию. Сначала мы добавим в список импортов библиотеки Apollo и компонент `Loading`:

```
import React from 'react';
import { View, Button, Text } from 'react-native';
import * as SecureStore from 'expo-secure-store';
import { useMutation, gql } from '@apollo/client';

import UserForm from '../components/UserForm';
import Loading from '../components/Loading';
```

Теперь можно добавить GraphQL-запрос:

```
const SIGNIN_USER = gql`
  mutation signIn($email: String, $password: String!) {
    signIn(email: $email, password: $password)
  }
`;
```

Далее нужно обновить функцию `storeToken` для сохранения строки токена, передаваемой в качестве параметра:

```
const storeToken = token => {
  SecureStore.setItemAsync('token', token).then(
    props.navigation.navigate('App')
  );
};
```

В конце мы оформим компонент как GraphQL-мутацию. Помимо этого, мы передадим несколько значений свойств в компонент `UserForm`, что позволит нам обмениваться данными мутации, определять тип вызываемой формы и использовать навигацию маршрутизатора.

```
const SignIn = props => {
  const storeToken = token => {
    SecureStore.setItemAsync('token', token).then(
      props.navigation.navigate('App')
    );
  };

  const [signIn, { loading, error }] = useMutation(SIGNIN_USER, {
    onCompleted: data => {
      storeToken(data.signIn)
    }
  });

  // В процессе загрузки возвращаем индикатор загрузки
  if (loading) return <Loading />;
  return (
    <React.Fragment>
      {error && <Text>Error signing in!</Text>}
      <UserForm
        action={signIn}
        formType="signIn"
        navigation={props.navigation}
      />
    </React.Fragment>
  );
};
```

Теперь можно внести небольшое изменение в компонент src/component/UserForm.js, которое позволит нам передавать введенную пользователем информацию в мутацию. Для этого мы обновим в компоненте функцию `handleSubmit`, чтобы передавать в мутацию значения формы:

```
const handleSubmit = () => {
  props.action({
    variables: {
      email: email,
      password: password
    }
  });
};
```

Написав мутацию и создав форму, мы добавили пользователям возможность авторизовываться в приложении, которое будет сохранять возвращаемый JSON Web Token для дальнейшего использования.

Аутентифицированные GraphQL-запросы

Теперь, когда наши пользователи могут авторизовываться в приложении, нам нужно использовать сохраняемый токен для аутентификации каждого запроса. Это позволит нам запрашивать пользовательские данные, например список заметок текущего пользователя или список заметок, отмеченных пользователем как избранные. Для реализации этого мы обновим конфигурацию Apollo, чтобы выполнять проверку существования токена и в случае его наличия отправлять значение этого токена с каждым вызовом API.

В файле src/Main.js сначала добавьте `SecureStore` в список импортов и обновите зависимости Apollo Client, включив в них `createHttpLink` и `setContext`:

```
// Импортируем библиотеки Apollo
import {
  ApolloClient,
  ApolloProvider,
  createHttpLink,
  InMemoryCache
} from '@apollo/client';
import { setContext } from 'apollo-link-context';
// Импортируем SecureStore для получения значения токена
import * as SecureStore from 'expo-secure-store';
```

Затем обновите конфигурацию Apollo Client для отправки значения токена с каждым запросом.

```
// Настраиваем  URI и кэш нашего API
const uri = API_URI;
const cache = new InMemoryCache();
const httpLink = createHttpLink({ uri });

// Возвращаем заголовки в контекст
const authLink = setContext(async (_, { headers }) => {
  return {
    headers: {
      ...headers,
      authorization: (await SecureStore.getItemAsync('token')) || ''
    }
  };
});

// Настраиваем Apollo Client
const client = new ApolloClient({
```

```
  link: authLink.concat(httpLink),
  cache
});
```

Настроив отправку токена в заголовке каждого запроса, мы можем обновить экраны `mynotes` и `favorites`, добавив запрос пользовательских данных. Если вы следовали инструкциям глав, посвященных веб-приложению, то эти запросы должны быть вам знакомы.

В src/screens/mynotes.js:

```
import React from 'react';
import { Text, View } from 'react-native';
import { useQuery, gql } from '@apollo/client';

import NoteFeed from '../components/NoteFeed';
import Loading from '../components/Loading';

// Наш GraphQL-запрос
const GET_MY_NOTES = gql`
  query me {
    me {
      id
      username
      notes {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;

const MyNotes = props => {
  const { loading, error, data } = useQuery(GET_MY_NOTES);

  // Если данные загружаются, приложение будет выводить сообщение о загрузке
  if (loading) return <Loading />;
  // Если при получении данных произошел сбой, выводим сообщение об ошибке
  if (error) return <Text>Error loading notes</Text>;
  // Если запрос выполнен успешно и содержит заметки, возвращаем их в ленту
  // Если же запрос выполнен успешно, но заметок не содержит, отображаем
  // сообщение
  if (data.me.notes.length !== 0) {
    return <NoteFeed notes={data.me.notes} navigation={props.navigation} />;
  } else {
```

```
    return <Text>No notes yet</Text>;
  }
};

MyNotes.navigationOptions = {
  title: 'My Notes'
};

export default MyNotes;
```

В src/screens/favorites.js:

```
import React from 'react';
import { Text, View } from 'react-native';
import { useQuery, gql } from '@apollo/client';

import NoteFeed from '../components/NoteFeed';
import Loading from '../components/Loading';

// Наш GraphQL-запрос
const GET_MY_FAVORITES = gql`
  query me {
    me {
      id
      username
      favorites {
        id
        createdAt
        content
        favoriteCount
        author {
          username
          id
          avatar
        }
      }
    }
  }
`;

const Favorites = props => {
  const { loading, error, data } = useQuery(GET_MY_FAVORITES);

  // Если данные загружаются, выдаем сообщение о загрузке
  if (loading) return <Loading />;
  // Если при получении данных произошел сбой, выдаем сообщение об ошибке
  if (error) return <Text>Error loading notes</Text>;
  // Если запрос выполнен успешно и содержит заметки, возвращаем их в ленту
  // Если запрос выполнен успешно, но заметок не содержит, отображаем
  // сообщение
  if (data.me.favorites.length !== 0) {
    return <NoteFeed notes={data.me.favorites} navigation={props.navigation} />;
```

```
    } else {
      return <Text>No notes yet</Text>;
    }
  };

  Favorites.navigationOptions = {
    title: 'Favorites'
  };

export default Favorites;
```

Рис. 24.4. Передача токена в заголовке каждого запроса позволяет выполнять в приложении индивидуальные пользовательские запросы

Теперь мы извлекаем индивидуальные пользовательские данные на основе значения токена, сохраненного на устройстве этого пользователя (рис. 24.4).

Добавление формы регистрации

На данный момент пользователь может входить в существующую учетную запись, но не может создавать ее. Стандартный UI-шаблон предусматривает

добавление ссылки на форму регистрации под ссылкой авторизации (или наоборот). Давайте добавим экран регистрации sign-up, чтобы пользователи могли создавать учетную запись из нашего приложения.

Для начала создадим новый компонент экрана src/screens/signup.js. Этот компонент будет почти идентичен экрану sign-in, но в нем мы будем вызывать GraphQL-мутацию `signUp` и передавать свойство `formType="signUp"` в компонент `UserForm`:

```
import React from 'react';
import { Text } from 'react-native';
import * as SecureStore from 'expo-secure-store';
import { useMutation, gql } from '@apollo/client';

import UserForm from '../components/UserForm';
import Loading from '../components/Loading';

// GraphQL-мутация signUp
const SIGNUP_USER = gql`
  mutation signUp($email: String!, $username: String!, $password: String!) {
   signUp(email: $email, username: $username, password: $password)
  }
`;

const SignUp = props => {
  // Сохраняем токен со значением ключа `token`
  // После сохранения токена переходим на главный экран приложения
  const storeToken = token => {
    SecureStore.setItemAsync('token', token).then(
       props.navigation.navigate('App')
    );
  };

  // Хук мутации signUp
  const [signUp, { loading, error }] = useMutation(SIGNUP_USER, {
    onCompleted: data => {
      storeToken(data.signUp);
    }
  });

  // Во время загрузки возвращаем индикатор загрузки
  if (loading) return <Loading />;

  return (
    <React.Fragment>
      {error && <Text>Error signing in!</Text>}
      <UserForm
        action={signUp}
        formType="signUp"
        navigation={props.navigation}
      />
```

```
    </React.Fragment>
  );
};

SignUp.navigationOptions = {
  title: 'Register'
};

export default SignUp;
```

Создав этот экран, мы можем добавить его в маршрутизатор. В файле src/screens/index.js сначала добавьте новый компонент в список импортов файлов:

```
import SignUp from './signup';
```

Затем обновите `AuthStack`, включив экран sign-up:

```
const AuthStack = createStackNavigator({
  SignIn: SignIn,
  SignUp: SignUp
});
```

После создания компонента и добавления его в маршрутизатор нам нужно еще добавить в компонент `UserForm` необходимые поля. Вместо создания компонента формы регистрации мы можем использовать свойство `formType`, которое передаем в `UserForm` для настройки формы в зависимости от типа.

В файле src/components/UserForm.js сначала мы обновим форму, чтобы она включала поле username, если `formType` равно `signUp`:

```
const UserForm = props => {
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();
  const [username, setUsername] = useState();

  const handleSubmit = () => {
    props.action({
      variables: {
        email: email,
        password: password,
        username: username
      }
    });
  };

  return (
    <FormView>
      <FormLabel>Email</FormLabel>
      <StyledInput
        onChangeText={text => setEmail(text)}
        value={email}
```

```
        textContentType="emailAddress"
        autoCompleteType="email"
        autoFocus={true}
        autoCapitalize="none"
      />
      {props.formType === 'signUp' && (
        <View>
          <FormLabel>Username</FormLabel>
          <StyledInput
            onChangeText={text => setUsername(text)}
            value={username}
            textContentType="username"
            autoCapitalize="none"
          />
        </View>
      )}
      <FormLabel>Password</FormLabel>
      <StyledInput
        onChangeText={text => setPassword(text)}
        value={password}
        textContentType="password"
        secureTextEntry={true}
      />
      <FormButton onPress={handleSubmit}>
        <ButtonText>Submit</ButtonText>
      </FormButton>
    </FormView>
  );
};
```

Затем добавим в нижнюю часть sign-in-формы ссылку, ведущую пользователя в форму sign-up:

```
return (
  <FormView>
    {/* здесь должен быть существующий код компонента формы */}
    {props.formType !== 'signUp' && (
      <TouchableOpacity onPress={() => props.navigation.navigate('SignUp')}>
        <Text>Sign up</Text>
      </TouchableOpacity>
    )}
  </FormView>
)
```

Теперь можно использовать стилизацию компонентов для изменения вида этой ссылки:

```
const SignUp = styled.TouchableOpacity`
  margin-top: 20px;
`;
const Link = styled.Text`
```

```
    color: #0077cc;
    font-weight: bold;
  `;
```

И в JSX компонента:

```
  {props.formType !== 'signUp' && (
    <SignUp onPress={() => props.navigation.navigate('SignUp')}>
      <Text>
        Need an account? <Link>Sign up.</Link>
      </Text>
    </SignUp>
  )}
```

В целом файл src/components/UserForm.js теперь будет выглядеть так:

```
  import React, { useState } from 'react';
  import { View, Text, TextInput, Button, TouchableOpacity } from 'react-
native';
  import styled from 'styled-components/native';

  const FormView = styled.View`
    padding: 10px;
  `;

  const StyledInput = styled.TextInput`
    border: 1px solid gray;
    font-size: 18px;
    padding: 8px;
    margin-bottom: 24px;
  `;

  const FormLabel = styled.Text`
    font-size: 18px;
    font-weight: bold;
  `;

  const FormButton = styled.TouchableOpacity`
    background: #0077cc;
    width: 100%;
    padding: 8px;
  `;

  const ButtonText = styled.Text`
    text-align: center;
    color: #fff;
    font-weight: bold;
    font-size: 18px;
  `;

  const SignUp = styled.TouchableOpacity`
```

```
  margin-top: 20px;
`;

const Link = styled.Text`
   color: #0077cc;
   font-weight: bold;
`;

const UserForm = props => {
  const [email, setEmail] = useState();
  const [password, setPassword] = useState();
  const [username, setUsername] = useState();

  const handleSubmit = () => {
    props.action({
      variables: {
        email: email,
        password: password,
        username: username
      }
    });
  };

  return (
    <FormView>
      <FormLabel>Email</FormLabel>
      <StyledInput
        onChangeText={text => setEmail(text)}
        value={email}
        textContentType="emailAddress"
        autoCompleteType="email"
        autoFocus={true}
        autoCapitalize="none"
      />
      {props.formType === 'signUp' && (
        <View>
          <FormLabel>Username</FormLabel>
          <StyledInput
            onChangeText={text => setUsername(text)}
            value={username}
            textContentType="username"
            autoCapitalize="none"
          />
        </View>
      )}
      <FormLabel>Password</FormLabel>
      <StyledInput
        onChangeText={text => setPassword(text)}
        value={password}
        textContentType="password"
        secureTextEntry={true}
```

```
        />
        <FormButton onPress={handleSubmit}>
          <ButtonText>Submit</ButtonText>
        </FormButton>
        {props.formType !== 'signUp' && (
          <SignUp onPress={() => props.navigation.navigate('SignUp')}>
            <Text>
              Need an account? <Link>Sign up.</Link>
            </Text>
          </SignUp>
        )}
      </FormView>
    );
};

export default UserForm;
```

После внесения всех этих изменений пользователь сможет не только авторизовываться в приложении, но и регистрировать новую учетную запись (рис. 24.5).

Рис. 24.5. Теперь пользователи смогут создавать учетную запись и перемещаться между экранами авторизации

Итоги

В этой главе мы рассмотрели процесс добавления в приложение аутентификации. С помощью комбинации элементов текстовой формы React Native, возможностей маршрутизации React Navigation, Expo-библиотеки SecureStore и GraphQL-мутаций мы можем создавать удобный для пользователя процесс аутентификации. Глубокое понимание этого типа аутентификации также позволяет нам изучать ее дополнительные React Native-методы, например AppAuth (https://oreil.ly/RaxNo) от Expo или GoogleSignIn (https://oreil.ly/Ic6BW). В следующей главе мы рассмотрим публикацию и распространение приложения React Native.

ГЛАВА 25

Дистрибуция мобильного приложения

Во времена моей учебы в старших классах еще в середине 90-х было очень модно загружать игры для графического калькулятора TI-81 (https://oreil.ly/SqOKQ). Кто-то один заполучал копию игры, а остальные с невероятным рвением начинали передавать ее друг другу, соединяя свой калькулятор с другим с помощью кабеля. За игрой на калькуляторе можно было коротать часы на задних партах класса или лекционного зала и при этом делать вид прилежно выполняющего учебное задание студента. Но описанный метод распространения игры, очевидно, был очень медленным: учащимся требовалось поддерживать проводное подключение в течение нескольких минут, в то время как остальным приходилось просто ждать. Сегодня цифровые карманные устройства намного превосходят калькуляторы того времени во многом благодаря тому, что мы можем с легкостью расширить их возможности через установку сторонних приложений.

Завершив начальную разработку нашего приложения, мы можем перейти к его дистрибуции, предоставив доступ к нему другим людям. В этой главе мы научимся настраивать файл app.js для распространения. Затем мы опубликуем приложение в открытом доступе в Expo, а в финале сгенерируем пакеты, которые можно будет отправить в онлайн-магазины Apple или Google Play.

Настройка app.json

Приложения Expo содержат файл конфигурации app.json, который используется для настройки специфических параметров приложения. Когда мы создаем новое приложение Expo, файл app.json автоматически генерируется для нас. Давайте взглянем на данный файл в нашем приложении:

```
{
  "expo": {
    "name": "Notedly",
```

```
    "slug": "notedly-mobile",
    "description": "An example React Native app",
    "privacy": "public",
    "sdkVersion": "33.0.0",
    "platforms": ["ios", "android"],
    "version": "1.0.0",
    "orientation": "portrait",
    "icon": "./assets/icon.png",
    "splash": {
      "image": "./assets/splash.png",
      "resizeMode": "contain",
      "backgroundColor": "#ffffff"
    },
    "updates": {
      "fallbackToCacheTimeout": 1500
    },
    "assetBundlePatterns": ["**/*"],
    "ios": {
      "supportsTablet": true
    },
    "android": {}
  }
}
```

Большая часть его элементов не требует пояснений, но я дополнительно опишу назначение каждого из них.

name

Имя приложения.

slug

URL для публикации приложения Expo по ссылке expo.io/project-owner/slug.

description

Описание проекта, которое будет использоваться при его публикации с помощью Expo.

privacy

Публичная доступность проекта Expo, которая может быть установлена как `public` или `unlisted`.

sdkVersion

Номер версии Expo SDK.

platforms

Платформы, для которых собирается приложение. Среди них могут быть `ios`, `android` и `web`.

version

Номер версии приложения, который должен соответствовать семантическим стандартам версионирования Semantic Versioning (https://semver.org).

orientation

Ориентация приложения по умолчанию. Может быть ограничена значениями `portrait` или `landscape` либо определяться вращением устройства пользователя при установке `default`.

icon

Путь к иконке приложения, используемый и для iOS, и для Android.

splash

Расположение изображения и настроек для экрана загрузки приложения.

updates

Конфигурация, определяющая проверку приложением обновлений «по воздуху» (over the air, OTA) после загрузки. С помощью параметра `fallBackToCacheTimeout` можно указывать продолжительность проверки в миллисекундах.

assetBundlePatterns

Позволяет указывать расположение материалов, которые должны быть включены в наше приложение.

ios and android

Активация настроек для конкретной платформы.

Конфигурация по умолчанию обеспечивает прочную основу для приложения. Однако есть ряд дополнительных настроек, которые можно найти в документации Expo (https://oreil.ly/XXT4k).

Иконки и экраны загрузки приложения

Маленькие квадратные иконки на наших устройствах стали одним из наиболее узнаваемых шаблонов в современном обществе. Уверен, что стоит вам закрыть глаза, как вы тут же сможете представить себе не один десяток иконок, вплоть до логотипа и конкретного цвета фона. Помимо этого, при нажатии пользователем на такую иконку возникает статическая заставка, которая отображается на время загрузки приложения. До этого момента мы использовали пустые иконки Expo и экран заставки по умолчанию. Однако все это можно заменить настраиваемым пользовательским дизайном.

Я добавил в каталог assets/custom вариант иконки Notedly и экрана заставки. Использовать их можно, либо заменив ими изображения в каталоге assets, либо указав в app.json путь к их файлам в субдиректории custom.

Иконки приложения

Файл icon.png — это квадратный файл формата PNG. Если мы укажем на него в свойстве `icon` файла app.json, Expo сгенерирует соответствующие размеры иконок для разных платформ и устройств. Изображение в данном случае должно быть именно квадратным и без прозрачных пикселей. Это простейший и самый понятный способ добавления иконок в приложение:

```
"icon": "./assets/icon.png",
```

Помимо одной кроссплатформенной иконки у нас есть вариант кастомизировать ее для конкретной платформы. Основная особенность данного подхода будет в добавлении раздельных стилей иконок для Android и iOS, особенно если вас интересует использование адаптивных иконок Android (https://oreil.ly/vLC3f).

В случае с iOS следует продолжить использовать один png-файл 1024 × 1024. В app.json:

```
"ios": {
  "icon": IMAGE_PATH
}
```

При желании использовать адаптивную иконку для Android нужно указать `foregroundImage`, `backgroundColor` (или `backgroundImage`), а также резервную статическую `icon`:

```
"android": {
  "adaptiveIcon": {
    "foregroundImage": IMAGE_PATH,
    "backgroundColor": HEX_CODE,
    "icon": IMAGE_PATH
  }
}
```

Мы же в нашем случае можем продолжить использовать одну статическую иконку.

Экраны-заставки

Экран-заставка — это полноэкранное изображение, которое будет кратковременно отображаться во время загрузки приложения на устройстве. Мы можем заменить предустановленное изображение Expo на одно из расположенных в каталоге assets/custom. Несмотря на то что размеры устройств различаются как внутри одной платформы, так и между ними, я решил оттолкнуться от ре-

комендаций Expo (https://oreil.ly/7a-5J) и использовать размер 1242 × 2436. После этого Expo изменит размер изображения, чтобы оно работало на разных экранах устройств, в том числе с разными соотношениями сторон.

Можно настроить наш экран-заставку в файле app.json так:

```
"splash": {
  "image": "./assets/splash.png",
  "backgroundColor": "#ffffff",
  "resizeMode": "contain"
},
```

По умолчанию мы устанавливаем белый фон, который может быть виден либо при загрузке изображения, либо, в зависимости от `resizeMode`, как рамка вокруг самого изображения экрана-заставки. Можно обновить следующее свойство для соответствия цвету нашего экрана:

```
"backgroundColor": "#4A90E2",
```

С помощью свойства `resizeMode` можно определить, как изображение будет подстраиваться под различные размеры экранов. Установив для него значение `contain`, мы сохраняем соотношение сторон исходного изображения. При использовании `contain` на некоторых размерах экранов и разрешениях `backgroundColor` будет видна рамка вокруг изображения экрана-заставки. Второй вариант — это установка `resizeMode` как `cover`, ведущая к расширению изображения на весь экран. Поскольку в нашем приложении есть небольшой градиент, давайте используем второй вариант:

```
"resizeMode": "cover"
```

На этом настройка иконки и изображение экрана-заставки завершена (рис. 25.1). Теперь можно рассмотреть варианты распространения приложения, чтобы сделать его доступным для других пользователей.

Рис. 25.1. Экран-заставка нашего приложения

Публикация через Expo

В процессе разработки приложение доступно для нас в Expo Client на физическом устройстве через нашу локальную сеть. Это означает, что мы можем к нему обращаться, пока компьютер, где ведется разработка, и телефон находятся

в одной сети. Expo дает возможность опубликовать наш проект, выгрузив приложение в Expo CDN и присвоив ему публичный URL. Таким образом оно станет доступно для всех желающих через Expo Client. Этот способ эффективен при тестировании или при необходимости быстрого распространения приложения.

Для публикации проекта можно нажать на ссылку Publish or republish project в Expo Dev Tools браузера (рис. 25.2) либо ввести `expo publish` в терминале.

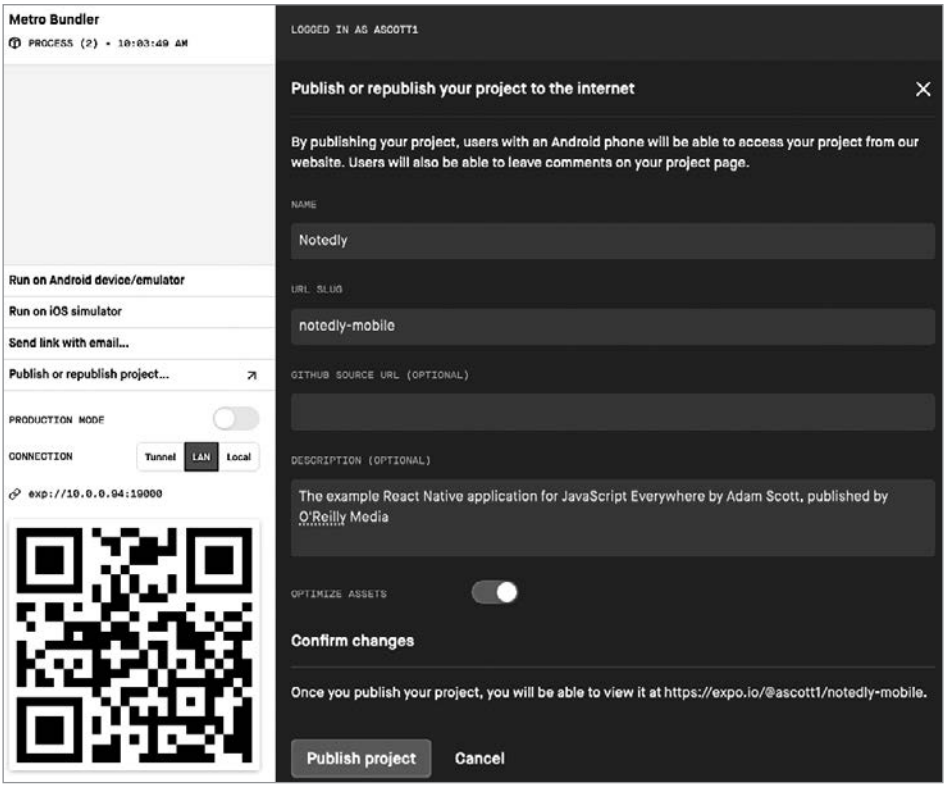

Рис. 25.2. Можно опубликовать приложение напрямую из Expo Dev Tools

После завершения сборки любой желающий сможет получить доступ к приложению через Expo Client, воспользовавшись ссылкой http://exp.host/@<username>/<slug>.

Создание нативных сборок

Несмотря на то что распространение приложений напрямую через Expo — это отличный вариант для его тестирования или скорейшего использования, нас

будет больше интересовать распространение через Apple App Store или Google Play Store. Для этого потребуется собирать файлы и выгружать их в соответствующий магазин.

ПОЛЬЗОВАТЕЛИ WINDOWS

Согласно документации Expo, пользователям Windows потребуется активировать подсистему для Linux (WSL). Это можно сделать, следуя руководству по установке для Windows 10 от Microsoft (https://oreil.ly/B8_nd).

iOS

Для генерации iOS-сборки требуется членство в программе разработчиков Apple (https://oreil.ly/E0NuU), которое стоит $99 в год. Имея необходимую учетную запись, мы можем добавить в файл app.json `bundleIdentifier` для iOS. Этот идентификатор должен соответствовать обратной DNS-записи:

```
"expo": {
"ios": {
    "bundleIdentifier": "com.yourdomain.notedly"
  }
}
```

Обновив app.json, мы можем перейти к генерации сборки. Находясь в терминале, введите из корня директории проекта следующее:

```
$ expo build:ios
```

После выполнения сборки вам будет предложено авторизоваться, используя ваш Apple ID. Следом за авторизацией вам зададут несколько вопросов, касающихся способа обработки учетных данных. Expo способен обрабатывать все учетные данные и сертификаты за нас, что можно разрешить, выбрав первый вариант из следующих:

```
? How would you like to upload your credentials? (Use arrow keys)
  («Как вы хотите выгрузить учетные данные? (Используйте клавиши стрелок)»)
□ Expo handles all credentials, you can still provide overrides
  («Предоставить обработку учетных данных Expo с возможностью их
  самостоятельного переопределения»)
  I will provide all the credentials and files needed, Expo does limited
  validat ion («Самостоятельно предоставить все учетные данные и необходимые
  файлы, Expo выполнит лишь их ограниченную проверку»)

? Will you provide your own Apple Distribution Certificate? (Use arrow keys)
  («У вас есть собственный сертификат распространения Apple? (Используйте
  клавиши стрелок)»)
□ Let Expo handle the process («Позволить Expo обработать процесс»)
  I want to upload my own file («Загрузить собственный файл»)
```

```
? Will you provide your own Apple Push Notifications service key? (Use arrow
  keys) («У вас есть собственный служебный ключ push-уведомлений Apple?
  (Используйте клавиши стрелок)»)
□ Let Expo handle the process («Позволить Expo обработать процесс»)
  I want to upload my own file («Загрузить собственный файл»)
```

Если у вас есть действующий аккаунт программы разработчиков Apple, Expo сгенерирует файл, который можно будет отправить в Apple App Store.

Android

Для Android можно сгенерировать либо пакетный файл (APK), либо файл сборки приложения Android App Bundle (AAB). AAB-файлы являются более современным форматом, поэтому давайте используем их. Если вам интересно, в документации по разработке для Android (https://oreil.ly/mEAlR) дано подробное описание преимуществ этого вида сборки, App Bundles.

Прежде чем начать, давайте обновим файл app.json, включив Android-идентификатор package. Как и в случае с iOS, им должна быть обратная DNS-запись:

```
"android": {
  "package": "com.yourdomain.notedly"
}
```

Теперь мы можем сгенерировать сборку приложения из терминала. Используйте cd для перехода в корневой каталог проекта и выполните следующее:

```
$ build:android -t app-bundle
```

При сборке приложений требуется делать их подпись. Хотя мы и можем сгенерировать эту подпись самостоятельно, Expo способен управлять хранилищем ключей за нас. После выполнения команды генерации сборки вы увидите следующее системное обращение:

```
? Would you like to upload a keystore or have us generate one for you? If
  you don't know what this means, let us handle it! :) («Хотите выгрузить
  хранилище ключей или нам сгенерировать его за вас? Если вы не понимаете,
  о чем речь, оставьте это нам! :)»)

1) Let Expo handle the process! («Позволить Expo обработать процесс!»)
2) I want to upload my own keystore! («Я хочу выгрузить собственное хранилище
   ключей!»)
```

При выборе первого варианта Expo сгенерирует сборку приложения за вас. В конце этого процесса вы сможете скачать файл и затем выгрузить его в Google App Store.

Дистрибуция через магазины приложений

Из-за постоянно меняющихся правил проверки и связанных с этим расходов я не стану рассматривать специфику отправки приложения в Apple App Store или Google App Store. В документации Expo (https://oreil.ly/OmGB2) собраны актуальные ресурсы и инструкции по управлению процессом рспространения приложений через указанные магазины.

Итоги

В этой главе мы рассмотрели публикацию и распространение приложений React Native. Инструменты Expo позволяют нам быстро публиковать приложения для тестирования и генерировать производственные сборки, которые можно выгружать в магазины приложений. Кроме того, Expo предоставляет на выбор разные степени контроля управления сертификатами и зависимостями.

На этом мы успешно завершили создание и публикацию бэкенд-данных API, а также веб-, десктопного и кроссплатформенного мобильных приложений!

Послесловие

У нас в Соединенных Штатах принято вручать выпускникам старшей школы копию книги доктора Сьюза «Это только начало!».

> Поздравляю! Сегодня ваш день. Вы отправляетесь в Удивительные Места! В путь!

Если вы дочитали книгу до конца, то нелишним будет позволить себе небольшой выпускной праздник. Мы рассмотрели очень много материала, начиная с построения GraphQL API с помощью Node и реализации серии разных UI клиентов. Однако это лишь малая толика ваших возможностей. Каждая из пройденных тем сама по себе охватывает целые книги и раскрывается в бесчисленном множестве онлайн-уроков. Я надеюсь, что вместо ощущения переполнения информацией у вас возникло чувство достаточной подготовленности к более глубокому изучению интересующих вас материалов и созданию удивительных проектов.

JavaScript — это «маленький» язык программирования, который «смог». Из скромного «игрушечного языка» он вырос в самый популярный инструмент программирования в мире. В итоге умение писать программы на JavaScript можно назвать суперсилой, позволяющей нам создавать практически любой вид приложений для какой угодно платформы. Учитывая, что мы говорим о суперсиле, считаю необходимым также добавить к этому распространенное выражение (https://oreil.ly/H02ca):

> «...чем больше сила — тем выше ответственность!»

Технология может и должна быть силой на службе добра. Я надеюсь, что вы способны применить полученные в этой книге знания для совершенствования мира. Это может подразумевать получение новой работы или реализацию стороннего проекта для повышения качества жизни как вашей семьи, так и окружающих, или же обучения других людей новым навыкам. Что бы это в итоге ни было, от применения знаний в благих целях выигрывают все.

Прошу вас не оставаться в стороне. Я буду рад получить любую информацию о создаваемых вами программах. Можете смело отправлять мне отзывы на адрес adam@jseverywhere.io или вступить в сообщество Spectrum (https://spectrum.chat/jseverywhere). Благодарю за чтение.

Адам

ПРИЛОЖЕНИЕ А

Локальное выполнение API

Если вы прочли только часть, посвященную UI, но не главы о разработке API, то вам все равно понадобится копия API, запущенная локально.

Сначала нужно убедиться, что у вас установлен MongoDB, как описано в главе 1. Настроив базу данных, вы можете скопировать API вместе с итоговым кодом. Для копирования кода на локальную машину откройте терминал, перейдите в директорию, где хранятся проекты, и выполните `git clone` для репозитория. Если вы этого еще не сделали, то также пригодится создать директорию notedly для организованного хранения кода всего проекта:

```
$ cd Projects
# Выполните команду mkdir, только если у вас еще нет директории notedly
$ mkdir notedly
$ cd notedly
$ git clone git@github.com:javascripteverywhere/api.git
$ cd api
```

После этого вам понадобится обновить переменные среды, сделав копию файла .sample.env и заполнив информацией новый .env-файл.

Выполните в терминале следующее:

```
$ cp .env.example .env
```

Теперь обновите значения .env-файла в текстовом редакторе:

```
## Database
DB_HOST=mongodb://localhost:27017/notedly
TEST_DB=mongodb://localhost:27017/notedly-test

## Authentication
JWT_SECRET=YOUR_PASSWORD
```

В конце можете запустить API, выполнив в терминале:

```
$ npm start
```

После выполнения этих инструкций вы получите копию API Notedly, выполняемую локально в вашей системе.

ПРИЛОЖЕНИЕ Б

Локальное выполнение веб-приложения

Если вы решили прочесть только часть, посвященную Electron, но не главы о веб-разработке, то вам все равно потребуется копия локально выполняемого веб-приложения.

Сначала нужно убедиться, что у вас есть локальная копия API. Если вы еще этого не сделали, обратитесь к приложению А для его настройки.

Настроив API локально, вы можете сделать копию веб-приложения. Для клонирования кода на локальную машину откройте терминал, перейдите в директорию, где хранятся проекты, и выполните **`git clone`** для репозитория проекта:

```
  $ cd Projects
  # Если вы храните проекты в директории notedly, перейдите в нее, используя
команду cd
  $ cd notedly
  $ git clone git@github.com:javascripteverywhere/web.git
  $ cd веб
```

Далее вам понадобится обновить переменные среды, сделав копию файла .sample.env и заполнив информацией новый .env-файл.

Выполните в терминале следующее:

```
  $ cp .env.example .env
```

Теперь обновите значения .env-файла в текстовом редакторе, чтобы он соответствовал URL локально выполняемого API. Если все было сохранено по умолчанию, то вам не потребуется вносить никаких изменений.

```
  API_URI=http://localhost:4000/api
```

Наконец, вы можете запустить финальный пример веб-кода. Выполните в терминале следующее:

```
  $ npm run final
```

После выполнения этих инструкций вы создадите копию веб-приложения Notedly, выполняемую локально в вашей системе.

Об авторе

Адам Д. Скотт — руководитель инженерного отдела, веб-разработчик и преподаватель, проживающий в Коннектикуте. В настоящее время он работает ведущим веб-разработчиком в Бюро финансовой защиты потребителей, где совместно с командой талантливых инженеров занимается созданием открытых веб-приложений. Кроме того, он более десяти лет работал в сфере образования, где занимался как обучением, так и составлением учебных планов по различным техническим тематикам. Он является автором книги «Wordpress for Education» (Packt, 2012), видеокурса «The Introduction to Modern Front-End Development» и серии докладов «The Ethical Web Development» (O'Reilly, 2016–2017).

Об обложке

Птица на обложке книги «Разработка на JavaScript» — это бронзовокрылый голубь (*Phaps calcoptera*), один из самых распространенных членов семейства голубиных, проживающих в Австралии. Эти птицы встречаются по всему континенту в различных средах обитания, где чаще всего их можно увидеть за поиском упавших на землю семян акации.

Бронзовокрылые — очень осторожные птицы: при малейшем беспокойстве они начинают громко хлопать крыльями, отступая к ближайшему дереву мульга (*Acacia aneura*). В ясный день солнечный свет проявляет бурово-зеленые пятна на их оперении. Мужских особей выделяет желто-белый лоб и розовая грудка, в то время как у самок окрас лба и грудки светло-серый. Тем не менее и у тех и у других есть выраженная белая линия, изгибающаяся из-под глаз в направлении задней части затылка.

Гнезда этих птиц достигают размера около 25 сантиметров в ширину и 10 сантиметров в глубину. Этого вполне достаточно, чтобы вместить два гладких белых яйца, которые откладываются одновременно. Примерно через 14–16 дней, на протяжении которых оба родителя высиживают эти яйца, из них вылупляются птенцы. В отличие от большинства птиц, за кормление новорожденных отвечают оба родителя. Для этого они выделяют похожую на молоко субстанцию из зоба — мышечной сумки, расположенной возле горла и служащей для хранения пищи.

Несмотря на то что статус бронзовокрылых голубей на сегодняшний день определен как вызывающий наименьшее беспокойство, многие из животных с обложек книг O'Reilly относятся к вымирающим видам. Все они очень важны для мира.

Иллюстрация обложки выполнена Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из книги *Lydekker's Royal Natural History*.