O'REILLY®

Фулстек JavaScript

Секреты, которые должен знать каждый мидл

SPRINT book

Милесия МакГрегор

Full-Stack JavaScript Strategies

The Hidden Parts Every Mid-Level Developer Needs to Know

Milecia McGregor

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Фулстек JavaScript

Секреты, которые должен знать каждый мидл

Милесия МакГрегор

SPRINT book 2026

ББК 32.988.02-018
УДК 004.738.5
М15

МакГрегор Милесия

М15 Фулстек JavaScript: Секреты, которые должен знать каждый мидл. — Астана: «Спринт Бук», 2026. — 432 с.: ил.

ISBN 978-601-12-7537-8

Как практикующий разработчик ПО вы уже умеете качественно выполнять задачи — на фронтенде или бэкенде. Пора перейти на следующую ступеньку карьерной лестницы и развить навыки, которыми обладают эксперты и senior-разработчики.

Милесия МакГрегор поможет разобраться, как работает вся система и как senior-разработчики принимают технические решения.

Вы изучите все необходимое для создания фулстек веб-приложения, развернутого на облачной платформе, поймете, как выявлять источники бизнес-логики для любого приложения, научитесь выбирать между различными архитектурами, сторонними сервисами и инструментами лучшие для создания масштабируемого проекта, освоите тонкие, но критически важные навыки senior-разработчика, включая стратегии взаимодействия с командами и анализ продуктовых решений.

Эта книга задумана как справочник, который можно открыть на любом этапе SDLC и использовать как чек-лист. Вы освоите все ключевые аспекты разработки приложений, чтобы уверенно принимать решения при возникновении вопросов, и будете понимать логику, сроки и методы принятия технических решений, а также эволюцию бизнес-требований.

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

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098122256 англ.

Authorized Russian translation of the English edition of Full Stack JavaScript Strategies ISBN 9781098122256 © 2025 The Milecia C. McGregor Living Trust. 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 978-601-12-7537-8

© Перевод на русский язык ТОО «Спринт Бук», 2026
© Издание на русском языке, оформление ТОО «Спринт Бук», 2026

Краткое содержание

https://t.me/it_boooks/2

Часть I
Начало нового проекта

Часть II
Разработка бэкенда

Часть III
Разработка фронтенда

Часть IV
Развертывание фулстек-приложения

Оглавление

Часть II
Разработка бэкенда

Часть III
Разработка фронтенда

Часть IV
Развертывание фулстек-приложения

От издательства

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научном редакторе русского издания

Владислав Краснолуцкий в Full-stack разработке более семи лет. За плечами десятки проектов: от архитектуры с нуля до поддержки и масштабирования уже существующих решений.

Рецензия научного редактора

Эта книга — находка для разработчиков с реальным боевым опытом. Она собирает воедино все те процессы, через которые мы проходим в работе: от выбора стека до выстраивания командных процессов. Особенно ценно, что автор не упускает нюансы, которые обычно остаются за кадром, — те самые детали, о которых не пишут в туториалах, но которые всплывают в реальных проектах.

После прочтения у меня сложилась цельная картина процесса разработки. Теперь книга стоит под рукой как практический справочник: когда стартует новый проект и нужно быстро освежить в памяти best practices по выбору технологий и организации процессов — она незаменима. Рекомендую всем, кто хочет систематизировать свой опыт и не изобретать велосипед в критические моменты.

Важно: книга охватывает впечатляющий спектр тем, но намеренно держит баланс между широтой и глубиной. Если вы уже сталкивались с этими концепциями — она отлично освежит память и разложит все по полочкам. А если какая-то тема окажется новой — книга даст прочный фундамент и четкий вектор для дальнейшего погружения. Считайте эту книгу вашим компасом, а не энциклопедией на 1000 страниц.

Влад Краснолуцкий

Предисловие

Цель этой книги — дать вам справочник (своего рода проверку адекватности) для работы как с новыми, так и с унаследованными проектами во фронтенде, бэкенде и при развертывании. Некоторые вопросы актуальны для обоих типов проектов: тестирование, производительность и безопасность. Многие приложения имеют общие базовые принципы, применимые в любой отрасли. В те моменты, когда вы ловите себя на мысли: «Почему я никогда об этом не слышал?», эта книга поможет вам уверенно задавать даже «простые» вопросы.

Для кого эта книга

Эта книга для вас, если вы хотите понять, как senior-разработчики словно по волшебству разбираются во всем и быстро осваивают сложные концепции, — я покажу, как это происходит.

Скорее всего, вы уже несколько лет работаете разработчиком: умеете писать надежный код для задач как на фронтенде, так и на бэкенде. Хотя у вас есть опыт full-stack разработчика, вы, вероятно, специализируетесь на одной его части. Например, на фронтенде вы знакомы с адаптивной верстkой, запросами к API и фреймворками вроде React, Astro или Svelte. На бэкенде — делали миграции БД, создавали API и настраивали простые схемы аутентификации.

Вы также умеете работать с Git (GitHub, GitLab) и тестировать изменения. Возможно, вы годами работали над одним проектом или переключались между разными, но в любом случае ваш опыт укладывается в описанные выше рамки.

Теперь вы готовы перейти на следующий уровень: разобраться, как устроена система в целом и почему принимаются те или иные технические решения. Именно об этом и пойдет речь в книге.

Чему вы научитесь

В этой книге вы изучите все необходимое для создания фулстек веб-приложения, развернутого на облачной платформе. Вы поймете, как выявлять источники бизнес-логики для любого приложения, научитесь выбирать между различными архитектурами, сторонними сервисами и инструментами лучшие для создания маштабируемого проекта. Вы освоите тонкие, но критически важные навыки senior-разработчика, включая стратегии взаимодействия с командами и анализа продуктовых решений. Независимо от вашего бэкграунда (фронтенд или бэкенд), вы изучите принципы проектирования и разработки, а также научитесь применять

их на практике в рамках жизненного цикла разработки ПО (SDLC, software development lifecycle) при создании продакшен-приложения.

Эта книга задумана как справочник, который можно открыть на любом этапе SDLC и использовать как чек-лист. Вы освоите все ключевые аспекты разработки приложений, чтобы уверенно принимать решения при возникновении вопросов. После прочтения вы будете понимать логику, сроки и методы принятия технических решений, а также эволюцию бизнес-требований.

Чего нет в этой книге

Эта книга — не углубленное руководство по конкретным инструментам и не учебник по JavaScript в целом. Хотя она охватывает широкий спектр тем с примерами, иллюстрирующими подходы уровня senior-разработчика, предполагается, что вы уже умеете читать код, искать ошибки и находить дополнительные учебные материалы.

Из-за большого охвата тем стратегии будут разбираться вместе с кодом. Эти стратегии — инструменты, которые можно применять в разных проектах, хотя они и неуниверсальны. Не существует единого подхода, работающего для любых двух проектов, ведь у каждого есть особенности. Поэтому цель — дать вам варианты на выбор в зависимости от ситуации.

Некоторые темы требуют более глубокого объяснения, чем может позволить глава или раздел. Ни одна книга не способна полностью раскрыть все нюансы, поэтому, хотя реализация некоторых концепций будет описана кратко, вы всегда найдете ссылки на дополнительные материалы для углубленного изучения.

Структура книги

Часть I «Начало нового проекта» будет относительно короткой и объяснит, как преобразовывать дизайн-макеты в задачи и вопросы к продуктовой команде и другим отделам. Здесь вы впервые познакомитесь с источниками бизнес-логики проекта.

Часть II «Разработка бэкенда» посвящена созданию серверной части проекта. Вы будете работать с NestJS (https://nestjs.com), разбирая ключевые аспекты разработки: безопасность, интеграцию сторонних сервисов и другие вопросы.

После готовности бэкенда вы перейдете к части III «Разработка фронтенда», где займетесь React-проектом. Вы создадите пользовательский интерфейс, уделив внимание типичным проблемам фронтенда: отзывчивости и производительности.

В части IV «Развертывание фулстек-приложения» вы детально изучите подключение фронтенда, бэкенда и вспомогательных систем для развертывания готового решения. К концу книги вы сможете уверенно входить в проект на любом этапе,

задавая правильные вопросы для уточнения задач и предлагая технические решения.

Вы заметите, что части книги различаются по объему. Это сделано намеренно, чтобы показать, на какие этапы разработки тратится больше всего времени. Некоторые компоненты проекта требуют больше усилий или имеют иной цикл обратной связи — книга стремится максимально точно отразить реальные условия работы.

Чтобы усилить практическую направленность этой книги, я включил мнения и размышления других инженеров-программистов в различные главы. Среди них: Итан Браун (Ethan Brown), основатель OptionLab и автор книг *«Learning JavaScript»* (3-е издание, O'Reilly)[1] и *«Web Development with Node and Express»* (2-е издание, O'Reilly)[2], а также Джефф Грэм (Jeff Graham), руководитель инженерного отдела Proxima AI с опытом в FinTech, EdTech, медиа и e-commerce.

Условные обозначения в книге

В книге используются следующие типографские обозначения.

Курсив

Курсивом выделены новые термины и важные понятия.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, пути, имена файлов и их расширений.

`Курсивный моноширинный шрифт`

Используется для оформления текста, который должен быть заменен значениями, предоставляемыми пользователем или определяемыми контекстом.

Шрифт без засечек

Используется для обозначения элементов интерфейса, папок, каталогов, URL и адресов электронной почты.

Этот элемент обозначает совет или рекомендацию.

[1] Браун Итан. Изучаем JavaScript. Руководство по созданию современных веб-сайтов.

[2] Браун Итан. Веб-разработка с применением Node и Express.

Этот элемент обозначает общее примечание.

Этот элемент предупреждает о важном моменте или потенциальной проблеме.

Использование исходного кода примеров

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресам: https://github.com/flippedcoder/dashboard-web и https://github.com/flippedcoder/dashboard-server.

При возникновении технических вопросов или проблем с использованием примеров кода пишите на support@oreilly.com.

Эта книга призвана помочь вам в работе. Как правило, примеры кода из книги можно свободно использовать в своих программах и документации без дополнительного разрешения, за исключением случаев:

- воспроизведения значительных частей кода;
- продажи или распространения примеров из книг O'Reilly;
- включения большого объема примеров в документацию к вашему продукту.

При этом цитирование книги и использование фрагментов кода в ответах на вопросы разрешено без ограничений.

Указание авторства приветствуется, но не является обязательным. Рекомендуемый формат ссылки:

«*Фулстек JavaScript: Секреты, которые должен знать каждый мидл* (*Full-Stack JavaScript Strategies: The Hidden Parts Every Mid-Level Developer Needs to Know*), автор Милесия Макгрегор (Milecia McGregor), издательство O'Reilly. Copyright 2025 Милесия Макгрегор, ISBN 978-1-098-12225-6».

Если применение вами примеров кода выходит за рамки добросовестного использования или указанных выше разрешений, обращайтесь на permissions@oreilly.com.

Благодарности

В первую очередь я хочу поблагодарить свою бабушку, Кэрол Мэтьюз (Carol Matthews), за то, что она на протяжении последних двух лет терпеливо слушала мои

бесконечные жалобы о трудностях написания этой книги. Каждый раз, когда мне нужно было выговориться, она была рядом.

Большое спасибо всем, с кем я работала в O'Reilly, — благодаря вам этот процесс был таким гладким. Вирджиния Уилсон (Virginia Wilson), без тебя я бы не справилась со всеми правками и отзывами! Аманда Куинн (Amanda Quinn), спасибо за организацию всех закулисных процессов и координацию с командой! Огромная благодарность Итану Брауну за детальные и вдумчивые замечания, которые ты давал мне в процессе работы над книгой. Твои комментарии и ресурсы, которыми ты поделился, стали для меня бесценными. Также спасибо Адаму Скотту (Adam Scott) за все предложения и обратную связь по книге.

Благодарю свою сестру, Сольей Гиббс (Soleil Gibbs), и близких друзей за то, что поддерживали меня, когда я выдыхалась или сталкивалась со сложными разделами. Вы помогали мне двигаться вперед, даже когда совсем не было сил. Отдельный источник мотивации — мой муж, который давал мне необходимое время и пространство для движения вперед.

В этом путешествии я была не одна, и я безмерно благодарна каждому, кто участвовал в нем — и в большом, и в малом.

Часть I

Начало нового проекта

ГЛАВА 1

Старт проекта

https://t.me/it_boooks/2

Добро пожаловать в команду! Скорее всего, вы, как и большинство читателей этой книги, — фронтенд- или бэкенд-разработчик, стремящийся перейти на уровень senior фулстек-разработчика. Представьте, что вы уже достигли этой цели: теперь вы — опытный фулстек-разработчик на JavaScript. Это означает, что в ваши обязанности входит участие в формировании технического направления проекта и координация работы разных команд, по окончании которой продукт попадет к пользователям. Ваша задача — начать разработку *greenfield-приложения* (то есть проекта с нуля, без наследуемого кода) на основе дизайнов и согласований с несколькими командами.

В этой главе вы узнаете:

- как взаимодействовать с несколькими командами для полного определения функционала;
- как декомпозировать дизайн на небольшие выполнимые задачи;
- как определить, какие данные вам необходимы для работы.

Проект, который вы будете создавать

Проект, который вы будете создавать в процессе изучения книги, представляет собой платформу электронной коммерции с доступом к большому объему пользовательских данных, включая персональные данные, и интеграцией с несколькими сторонними сервисами. Вашей команде поручено создать пользовательскую панель управления, с помощью которой клиенты могут совершать покупки, просматривать историю заказов и выполнять другие действия, а также взаимодействовать с цифровыми товарами (например, электронными книгами). Функциональность панели будет зависеть от уровня прав пользователя в приложении.

Вы начнете работу с новыми командами: продуктовой командой и командой дизайнеров. Продуктовая команда будет согласовывать функционал приложения со стейкхолдерами, а дизайнеры — проектировать интерфейсы и первичный UX.

На первом этапе команды совместно подготовят макеты экранов и документацию по поведению системы, которые будут представлены на вводном собрании.

Вводное собрание

На вводном собрании по запуску проекта (kickoff meeting) продуктовая команда обычно демонстрирует макеты, созданные дизайнерами. Хотя в процессе разработки эти макеты, скорее всего, будут меняться, они уже должны быть готовы примерно на 80 %. На этом этапе команда покажет вам пользовательские сценарии, объяснит логику взаимодействия с приложением и обозначит необходимые данные. Не стесняйтесь задавать много вопросов — при техническом анализе макетов часто выявляются аспекты, которые не учли ни продакт-менеджеры, ни дизайнеры.

Обязательно включите в программу вводного собрания краткое знакомство участников команды, чтобы все понимали, с кем будут сотрудничать. Хотя есть соблазн сразу перейти к рабочим вопросам, не забывайте, что установление личного контакта с коллегами не менее важно.

Помните, что весь этот процесс строится на совместной работе. Чем раньше и чаще вы будете задавать вопросы, тем легче пойдет проект. Итак, в нашем случае на первом этапе, чтобы обозначить концепцию, нужно создать всего два экрана, хотя продуктовая команда предупредила, что функциональность со временем расширится. Давайте рассмотрим эти экраны и пройдемся по пользовательскому сценарию.

Скриншоты сделаны в Figma (https://www.figma.com) — популярном инструменте для дизайна. Обычно где-то уже есть макет, который визуально описывает логику приложения. В процессе разработки вы будете последовательно фокусироваться на каждом из этих представлений, например на экране с информацией о пользователе (рис. 1.1).

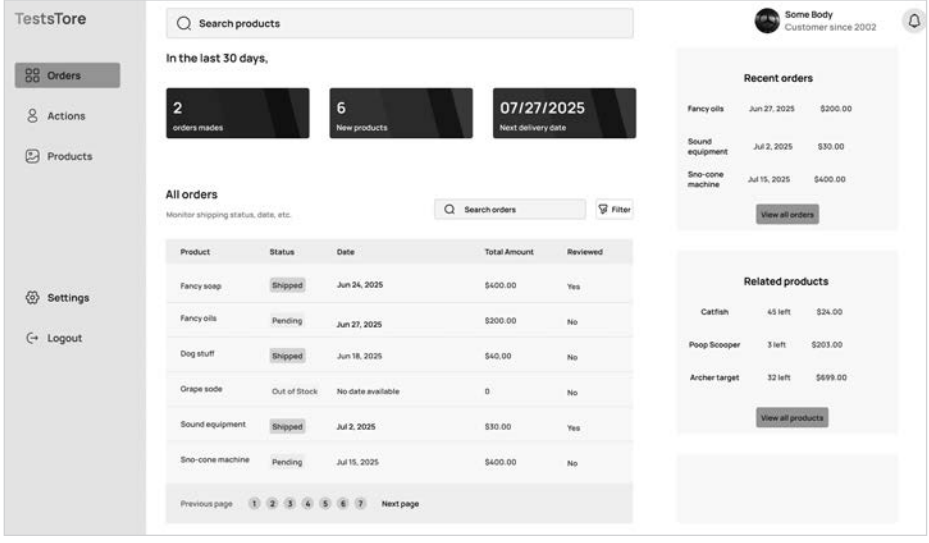

Рис. 1.1. Дизайн экрана с пользовательской информацией в Figma

На рис. 1.2 показан экран, где пользователи могут просматривать свою информацию: заказы, рекомендованные товары и другие данные. Здесь же они редактируют данные (например, адрес), отменяют заказы и просматривают цифровые покупки.

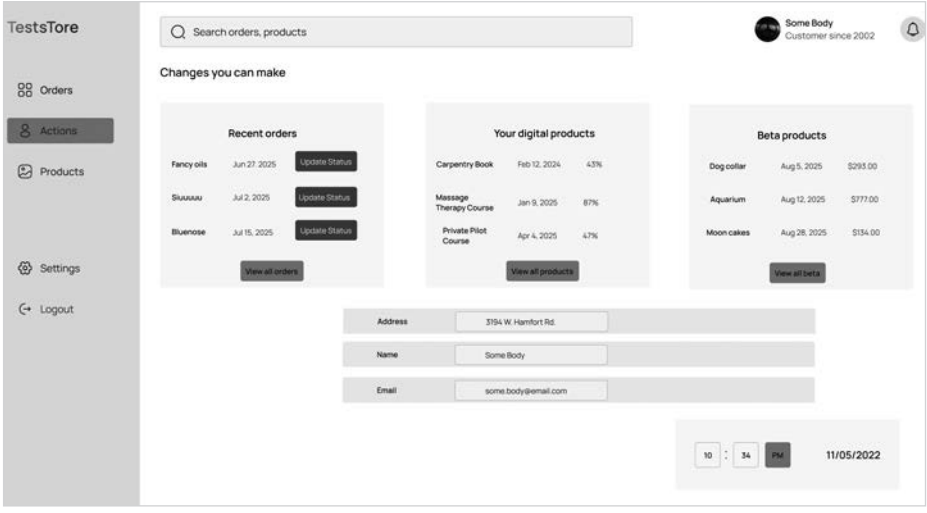

Рис. 1.2. Дизайн экрана пользовательских действий в Figma

На рис. 1.3 представлен мобильный адаптив этих экранов. Большинство проектов, создаваемых для внешних пользователей, требуют мобильной версии, поэтому, если ее нет в макетах, обязательно проясните этот момент!

Многие дизайнеры используют Figma при создании макетов приложения или отдельных функций. Среди других инструментов, которые вам могут встретиться, — InVision (теперь Miro, https://miro.com) или более новая альтернатива Penpot (https://penpot.app). В Figma-макетах есть возможность просмотреть сгенерированные HTML и CSS, что дает возможность разработчикам проверить размеры элементов при верстке. Однако его CSS не всегда полностью точен: используйте его как отправную точку, но обязательно дорабатывайте стили под требования проекта.

К моменту получения макетов продуктовая команда и команда дизайнеров уже проведут несколько встреч для проработки деталей. Но это не гарантирует полной готовности — здесь вступаете вы. Вам предстоит совместно с командой определить технический подход к реализации — опирайтесь на макеты и описание пользовательских сценариев.

Давайте разберем эти макеты так же, как вы будете делать это с продуктовой командой. На первом экране (рис. 1.1) представлены:

- боковая панель навигации со ссылками на другие разделы приложения;
- поисковая строка в верхней части страницы для поиска товаров;

- блок рекомендуемых товаров вместе с последним заказом пользователя;
- сортируемая таблица с постраничным выводом и несколькими колонками, отображающая все заказы пользователя.

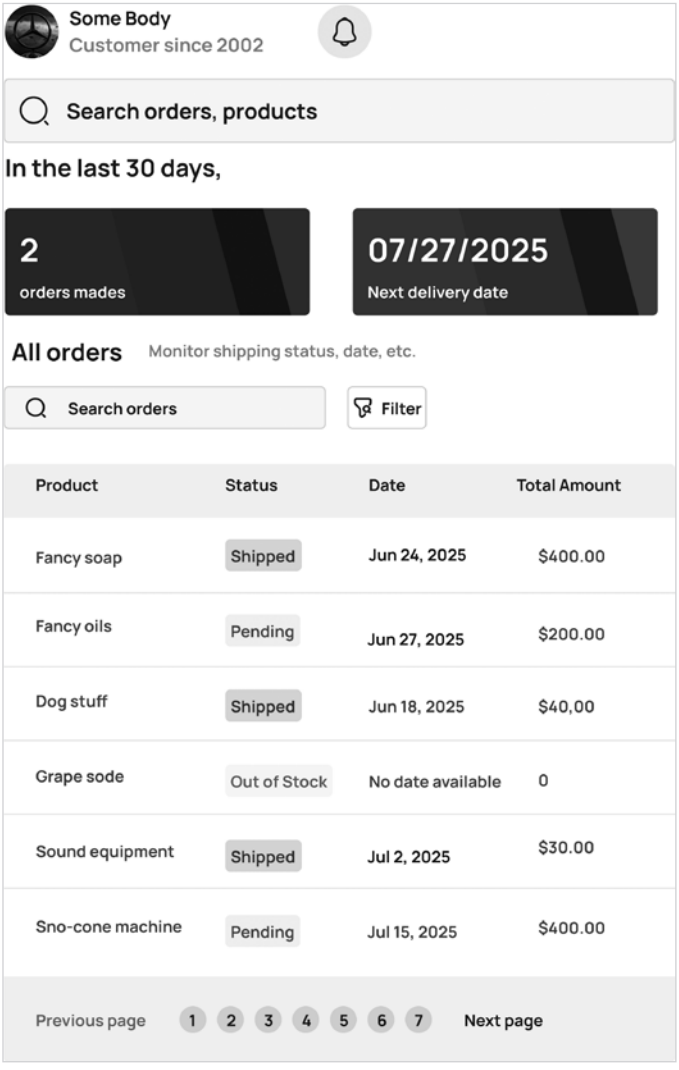

Рис. 1.3. Дизайн мобильного представления приложения в Figma

На втором экране (см. рис. 1.2) есть несколько секций с интерактивными полями. Пользователи могут переключать настройки, обновлять данные и выполнять действия на основе своих покупок. Этот экран сложнее первого, поэтому стоит

детально разобрать его вместе с командой. Продуктовая команда и дизайнеры часто будут обращаться к вам как к senior-разработчику за окончательным решением по техническим вопросам.

Никогда не давайте обещаний, не обсудив их с командой, — даже если на вас давят. Продуктовая команда может пригласить вас на встречу, чтобы оценить сложность реализации их идеи. В таких случаях часто ждут, что вы быстро согласитесь с предложенными сроками, и вам, как будущему senior-разработчику, нужно научиться уверенно с этим работать. Четко заявите, что не сможете дать точную оценку, пока не проконсультируетесь с командой. Коллеги-разработчики могут указать на проблемы, которые вы упустили, ведь в итоге они тоже будут работать над функционалом. Если продуктовая команда продолжает требовать дату — стойте на своем и не обещайте ничего, не обсудив с командой. Так вы избежите ситуации, когда ваши коллеги окажутся поставленными в жесткие временные рамки из-за вас.

После того как вы получили макеты и документацию по работе экранов, разберитесь в бизнес-логике приложения. Лучше это сделать до обсуждения проекта со своей командой. Четкое понимание причин, стоящих за дизайнерскими решениями, значительно упростит их объяснение другим разработчикам. Анализируя эти материалы, учитывайте ключевые аспекты, вытекающие из предоставленных макетов.

Вопросы проектирования

На этом этапе у вас неизбежно возникнет множество вопросов — это абсолютно нормально. Сейчас самое время погрузиться в детали и скрупулезно их проработать. Один из способов набраться опыта в технической оценке дизайна — познакомиться с разными дизайн-макетами. Например, команда дизайнеров, с которой вы сотрудничаете, вероятно, разрабатывает интерфейсы и для других приложений вашей компании.

Если вам непонятно, как должен работать выпадающий список, спросите, есть ли другие макеты для сравнения. Обычно они есть — изучите досконально, как они работают. Самые ценные вопросы часто рождаются при анализе разных решений. Еще один способ познакомиться с различными подходами — начать внимательно анализировать приложения, которыми вы пользуетесь. Задумывались ли вы, например, о пользовательском сценарии (user flow) в банковском приложении или в приложении для тренировок и медитации? Это реальные примеры продуктовых дизайнов, с которыми можно проводить сравнения.

Выделить время на исследовательскую работу бывает непросто. Один из способов — создать тикет, где будут подробно описаны область исследования, текущие мысли по проблеме и открытые вопросы. Добавьте также пункты для обсуждения, чтобы придать тикету более формальный вид.

Анализируйте, как реализованы дизайны в других продуктах и как ведут себя интерфейсы при детальном рассмотрении. Фиксируйте возникающие вопросы по мере выявления различий.

Важное замечание при составлении списка вопросов: прежде чем задавать вопрос, проверьте предоставленную документацию. Часто ответы уже есть в материалах, и их изучение сэкономит время всем. Выделите около часа на изучение инженерной документации по другим проектам команды, а также документации сторонних сервисов и пакетов, которые вы планируете использовать.

Возможно, вам интересно, какие вопросы стоит задавать. Это всегда зависит от дизайна и бизнес-логики проекта.

Вот стартовый список вопросов при анализе экранов нашего проекта:

- Нужна ли поддержка переводов на другие языки?
- Какие элементы брендбука (цвета, шрифты, адаптивные размеры) должны применяться в приложении?
- Какие права нужны пользователям для просмотра их информации?
- Какие разрешения требуются для действий в приложении?
- Сколько типов пользователей предусмотрено?
- Какова ожидаемая активность пользователей (для определения времени жизни токена доступа)?
- Какие данные пользователей мы будем обрабатывать?
- Нужно ли отображать эти данные для разных периодов времени?
- Должна ли таблица сортироваться по всем колонкам или только по определенным?
- По каким критериям определяются рекомендуемые заказы?
- Какие типы данных допустимы в полях ввода?
- Что показывать пользователю после сохранения изменений?
- Что показывать пользователю или какие действия предлагать выполнить, если возникает API-ошибка на стороне клиента?
- Нужна ли дополнительная аутентификация для операций с чувствительными данными (например, для изменения персональных данных)?
- Отображать ошибки ввода непосредственно в поле или при нажатии кнопки Submit?
- Нужна ли поддержка планшетов и других мобильных устройств?
- Будет ли бэкенд использоваться другими приложениями?

- Потребуется ли интеграция фронтенда в другие системы?
- Должно ли приложение подпадать под какие-либо комплаенсы, будет ли в отношении него проводиться аудит?

Как видите, список может быть очень длинным, и вы наверняка добавите свои вопросы. Не беспокойтесь о его объеме — редко удается проработать все сразу, поэтому расставьте приоритеты по важности для функционала. Лучше задать максимум вопросов на старте, а не когда пройдено полпути до дедлайна. По мелочам (например, обработка форм) можно полагаться на свой опыт.

Вопросы могут быть самыми разными, но продуктовая команда или команда дизайнеров должна уметь на них отвечать. Пока что здесь нет ничего технического, хотя ответы на эти вопросы будут определять технические решения. Своевременное получение ответов позволяет senior-разработчикам хорошо понять бизнес-логику.

Чтобы принимать оптимальные технические решения, необходимо понимать назначение функционала приложения и его роль в общем бизнес-контексте. Это поможет сделать приложение максимально поддерживаемым, поскольку вы будете знать вектор развития бизнеса. Частично это понимание формируется на основе данных, с которыми предстоит работать.

Дизайн на основе данных

Данные определяют дизайн. Вся работа дизайнеров строится вокруг оптимальной подачи данных и взаимодействия с ними. Например, экран с таблицей: данные не обязаны отображаться именно в табличном виде — это может быть сворачиваемый список или график. Поэтому необходимо детально разобраться в бизнес-логике и данных. Это одна из причин, по которой фронтенд-разработка обычно зависит от бэкенд-разработки. Подробнее об этом — в главе 2.

При анализе макетов полезно фиксировать данные, которые потребуются для каждого экрана. Например, на экране настроек в одном разделе могут понадобиться имя пользователя, адрес доставки и языковые предпочтения. Отмечайте предполагаемые имена переменных и типы данных — этой информацией вы поделитесь с командой, и именно с нее можно начать обсуждать реализацию проекта.

Также согласуйте с продуктовой командой корректность этих значений и ожидаемое поведение приложения. Всегда уточняйте у коллег правильность данных и их взаимосвязанность. Они не будут объяснять это техническими терминами, но проведут вас через различные сценарии, чтобы вы увидели общую картину.

Это напоминает фазу исследования проекта для вашей команды разработчиков. После анализа макетов и подготовки вопросов с заметками будьте готовы обсуждать конкретные функции.

Разбивка дизайнов на задачи

Вы завершили первый раунд уточнения проекта! Мы предполагаем, что к этому моменту продуктовая команда ответила на все вопросы и дизайны обновлены. Теперь вы будете совместно выделять функции из этой информации. Здесь вы впервые начнете использовать систему тикетов (Jira (https://oreil.ly/qmodA), Trello (https://trello.com/), ClickUp (https://clickup.com/) или Shortcut (https://www.shortcut.com/)). На рис. 1.4 показан пример системы тикетов в Trello.

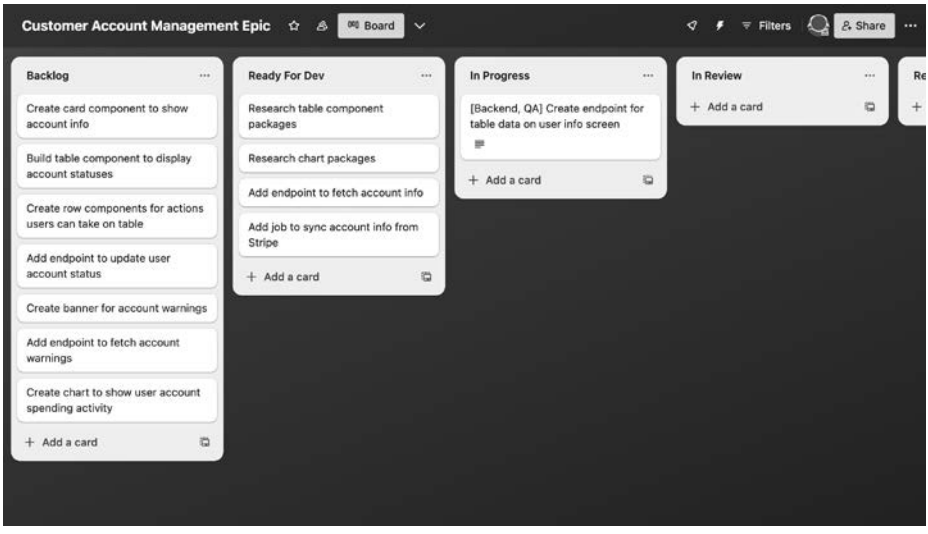

Рис. 1.4. Пример бэклога в Trello

Созданные тикеты будут уточняться по мере подключения других разработчиков и команд. Цель этапа — сформировать практически реализуемые задачи для элементов дизайна. Например, экран с пользовательской информацией содержит несколько компонентов (раздел с избранным, таблицу), которые можно выделить в отдельные функции. Ваша задача — стимулировать обсуждение того, как практически реализовать дизайн, постепенно переводя процесс в техническое русло.

Попробуйте группировать функции по связям между данными — это упростит внесение и развертывание изменений в бэкенде параллельно с разработкой фронтенда. На этом этапе еще можно выявить серьезные проблемы при проектировании стека технологий. Все идеи по функционалу или технические требования сразу оформляйте в тикеты — без тикета работа над ними не начнется.

Уточнение задач на основе требований к функционалу

На этом этапе продуктовая команда должна предоставить пользовательские сценарии (user stories), которые вам предстоит наполнить техническими деталями.

Эти сценарии — часть документации, определяющей разрабатываемый функционал. Обычно их пишут с точки зрения пользователя, чтобы раскрыть контекст взаимодействия с приложением. Хотя так бывает не всегда: часто вам придется совместно с продакт-менеджерами дорабатывать требования, опираясь на наработки предыдущего этапа.

Предположим, ваша команда работает по классическому Agile с двухнедельными спринтами и стандартными митингами: краткими ежедневными встречами (часто в формате стендапов), уточнением бэклога, планированием спринтов и ретро. Начните разбирать тикеты в Jira на встрече по уточнению бэклога. В ней должны участвовать разработчики, QA-тестировщики, дизайнеры и продакты — это позволит всем прояснить свои вопросы и озвучить замечания. Тикеты функционала нужно дробить на минимальные осмысленные задачи для разработчиков.

Здесь важно балансировать между строгими правилами и здравым смыслом. Например, «минимальным тикетом» может быть реализация эндпоинта со всеми сопутствующими подзадачами, а может быть — каждый вызов события. Ориентируйтесь на логику тестирования: какой минимальный блок функционала имеет смысл проверять целиком? По Agile и способам принимать подобные решения написаны целые книги, например, полезные материалы есть в статьях Atlassian (https://oreil.ly/chgWf).

Чтобы синхронизировать ожидания с возможностями команды, для каждого тикета оценивайте сложность. Не бойтесь задерживаться на обсуждении тикетов во время уточнения бэклога: если вы потратите много времени на один тикет, это четко укажет продуктологам на зоны роста, которые можно обсудить на ретро.

Убедитесь, что для каждого тикета есть критерии приемки, согласованные с продактами, QA и вашей командой. Для тикетов по пользовательскому интерфейсу (User Interface, UI) обязательно прикрепляйте макеты под десктоп и мобильные устройства. Добавляйте справку о конкретном ожидаемом функционале в приложении — это даст разработчикам контекст. Оценки сложности тикета должны быть едиными для всех членов команды.

Такая скрупулезность может показаться избыточной, но она экономит время команды. Обсуждение всех ключевых моментов со всеми участниками на старте помогает избежать блокировок в работе. При таком уровне детализации каждого тикета у вас не останется неоднозначностей в постановке задачи. Любой разработчик должен четко понимать, что нужно сделать (но не как делать — это определяется в ходе обсуждения в команде). Такие подробные тикеты инициируют содержательные технические дискуссии, которые влияют на архитектуру проекта.

На этом этапе команда разработки детализирует задачи от продакт-менеджеров. Полезно фиксировать в тикетах ключевые решения со встреч — это сохраняет контекст для код-ревью и тестирования. Например, если внешний сервис требует данные в специфичном формате, это станет основанием для создания

хелпер-функций, но их необходимость будет продиктована продуктовыми требованиями.

Хотя процесс кажется долгим, тикеты на следующий спринт обычно формируются за несколько дней. После его старта можно приступать к работе, но перед выбором тикетов и написанием кода потребуется еще несколько обсуждений.

Вот пример хорошо оформленного тикета:

Название: [Backend, QA] Создать эндпоинт для данных таблицы на экране информации о пользователе

Описание: как пользователь, я хочу видеть таблицу с историей покупок. Данные должны включать название товара, цену, предполагаемую дату доставки и количество в каждой строке. Таблица должна поддерживать сортировку по клику на заголовках столбцов и пагинацию для просмотра истории за последние 12 месяцев. Внешний вид — согласно приложенным макетам (см. рис. 1.1).

Критерии приемки:

- обновить схему данных для хранения истории покупок (название товара, цена, дата доставки, количество);
- применить миграцию БД;
- создать эндпоинт, возвращающий историю покупок пользователя за 12 месяцев;
- убедиться, что аутентификация работает корректно;
- реализовать валидацию входных данных;
- написать тесты для нового эндпоинта;
- составить инструкцию по использованию эндпоинта для команды фронтенда.

Оценка: 5 стори-пойнтов

Обсуждение сроков с продуктовой командой и другими отделами

На этом этапе проекта можно считать, что у вас есть четко сформулированные тикеты и максимально проработанные детали. Пришло время для обсуждения на более высоком уровне. После оценки всех тикетов продуктовая команда будет готова установить дату релиза.

Иногда продуктовая команда предлагает дату релиза до оценки тикетов. Если их сроки оказываются короче, чем требуется технической команде для качественной реализации, — возражайте. Всегда озвучивайте проблему, если понимаете, что сроки слишком нереалистичные, — это избавит всех от лишнего напряжения.

В обсуждении будет участвовать не только ваша команда. Возможно, потребуется согласование с разработчиками других проектов, командой DevOps, поддержкой и QA — это зависит от масштаба ваших изменений. Не фиксируйте дату релиза до переговоров с этими командами. Важно понять, как ваши цели соотносятся с их планами и как облегчить им работу.

Взаимодействие с другими командами разработчиков

Может оказаться, что для вашего проекта потребуется функционал стороннего кода, который пишет другая команда. Заранее уведомите ее и объясните детали, чтобы они включили задачу в свой спринт. Команды не всегда знают, когда функционал одного приложения пересекается с их продуктом. Четко объясните свои потребности, упростив формулировки. Укажите целевую дату релиза и узнайте, смогут ли они уложиться в эти сроки.

С другой стороны, новый функционал вашего приложения может вызвать критические изменения у других команд — предупредите их о необходимых доработках и сроках. Не стесняйтесь лишний раз проверить, повлияет ли ваша разработка на другие отделы. Если да, то фиксируйте подобные изменения в документации для пользователей.

Подготовьте перед обсуждением следующее.

Четкие технические требования.

Сформулируйте задачу так же детально, как для своей команды, и оставьте время на уточнения.

Предполагаемую область изменений в их коде.

Не нужно самим вносить правки, но обозначьте отправные точки.

Контекст запрашиваемых изменений.

Даже незначительное, на ваш взгляд, изменение может повлиять на критически важный функционал их приложения. Убедитесь, что коллеги понимают необходимость этих изменений.

Гибкий дедлайн.

Уточните сроки реализации, но учитывайте, что они могут двигаться.

Ресурсы для самостоятельного выполнения работы.

Если самостоятельное внесение изменений с последующим код-ревью ускорит процесс — рассмотрите этот вариант.

Когда эти вопросы с командами разработчиков будут решены, можно переходить к обсуждению с другими участниками. Порядок этих обсуждений неважен — главное, чтобы они состоялись. В первую очередь можно поговорить с теми, кто доступен в данный момент.

Координация с командой DevOps

При развертывании в разных средах потребуется согласование с командой DevOps (если таковая имеется). В небольших компаниях развертывание часто ложится на разработчиков. В этой книге предполагается, что у вас есть как минимум один DevOps-инженер, отвечающий за инфраструктуру и пайплайны.

Для нового приложения потребуется настроить инфраструктуру:

- тестовые среды для QA;
- продакшен-среду;
- другие необходимые среды.

Вы можете участвовать в настройке CI/CD (continuous integration and continuous deployment)-пайплайна через CircleCI (https://circleci.com), Jenkins (https://www.jenkins.io) или GitHub Actions (https://oreil.ly/bX9eg), но вопросы инфраструктуры пусть остаются за командой DevOps.

Команда DevOps предоставит ключевую информацию (например, об облачном провайдере), а вы сообщите им информацию об используемом стеке (языки, фреймворки). Общая цель — подготовить тестовую среду для разработки и тестирования до выпуска в продакшен. Раннее вовлечение команды DevOps упростит процесс разработки.

Помните: у всех команд, с которыми вы взаимодействуете, уже есть свои приоритеты. Поэтому учитывайте, что они могут не ответить вам сразу. Вот почему важно согласовать техническое направление с командой DevOps как можно раньше. Крайне нежелательно приводить DevOps в состояние цейтнота прямо перед важным дедлайном.

Будьте готовы быстро провести онлайн-встречу, чтобы обсудить проект и убедиться, что команда DevOps понимает ожидаемую производительность и нагрузку на приложение. В ходе созвона обсудите самое насущное.

Заранее разработайте плейбуки — стратегии и шаги для устранения простоев и других проблем в продакшене.

При возникновении инцидента бывает сложно быстро определить оптимальные действия, поэтому обсудите это с командой DevOps заранее.

Определите оптимальное количество сред для приложения.

В одних проектах достаточно трех сред, в других создают среды для каждой ветки Git. Найдите баланс между удобством поддержки инфраструктуры и гибкостью для тестирования.

Выберите стратегию развертывания.

Команда DevOps должна четко понимать, когда и как часто запускать скрипты. Часть процессов будет автоматизирована, часть — выполняться вручную. Идеальная стратегия — это та, которая устраивает обе команды.

Определите, кто поддерживает CI/CD.

Большинство инструментов используют YAML-файлы для настройки пайплайнов, и эти файлы хранятся в репозитории. Иногда правки вносят разработчики, иногда — DevOps или же обе команды. Главное — убедиться, что все понимают задачи одинаково.

Договоритесь о версиях среды выполнения JavaScript (например, Node.js, Deno или Bun), которые будут поддерживаться.

Серверы не всегда работают на последней версии среды выполнения, что может вызывать странные ошибки в приложении. Узнайте минимальную версию среды выполнения, с которой совместим ваш проект, на случай непредвиденных ситуаций.

Выясните сроки обновления учетных данных.

На некоторые приложения влияют ключи и секреты (например, npm-токены), которые управляются инфраструктурой. Добавьте тикет с напоминанием в бэклог, чтобы запланировать их обновление до истечения срока действия и избежать проблем в продакшене.

Определите место хранения ассетов.

Для одних приложений это будет часть облачного провайдера, для других — сторонний сервис. Даже если окончательное решение не принято, создайте тикет для возврата к этому вопросу позже.

Выделите время на совместное тестирование приложения перед развертыванием.

Вы лучше знаете ожидаемое поведение приложения, поэтому DevOps-инженеры будут полагаться на вас при проверке окружения. Они могут запустить приложение, но вам нужно убедиться, что оно работает корректно.

Заложите время на пост-развертывание.

Первые несколько развертываний почти всегда требуют исправлений. Занесите эту задачу в календарь — если все пройдет гладко и исправления не потребуются, у команды будут повод и время отпраздновать успех.

Важно помнить: общаясь с DevOps-инженерами, вы не обязаны понимать все технические детали их работы. Со временем вы вникнете в нюансы, а совместная работа с командой DevOps поможет быстрее разобраться в ее задачах. Если не хотите углубляться в DevOps — ничего страшного, они возьмут остальное на себя.

Далее нужно согласовать действия с командой QA.

Взаимодействие с командой QA

Даже senior-разработчики могут пропустить баги, поэтому QA-тестирование критично для защиты продакшена. Проблемы возникают из-за интеграционных сбоев, конфликтов в пул-реквестах, неучтенных пограничных случаев (edge cases) и других факторов. Своевременная передача команде QA новых функций и исправлений обеспечивает соблюдение сроков релиза. Вам необходимо отправить новый код QA-инженерам, дать им время на тестирование и создание отчета о дефектах, затем выделить время на исправление этих дефектов и передать код обратно в QA.

Возможна ситуация, когда отдельной команды QA вообще не будет, — это зависит от компании. В стартапах на ранней стадии часто нет выделенного отдела QA: тестированием занимаются разработчики, а иногда даже CEO. В других компаниях продуктовая команда может выполнять ручное тестирование функционала. Независимо от наличия команды QA, рекомендации из этого раздела помогут вам организовать более тщательное самостоятельное тестирование.

Этот цикл приемки-передачи кода может занимать значительное время. Чтобы сократить сроки и обеспечить приоритетное тестирование критичных тикетов, привлекайте QA-специалистов к обсуждениям разработчиков. Они задают вопросы о функционале и багах с точки зрения тестирования, которые важно учитывать. Написанные ими тест-кейсы фактически становятся историей приложения: большинство функций получает тестовое покрытие (автоматизированное или ручное), причем эти тесты сохраняются даже при удалении самого функционала, так как команда QA документирует изменения иначе, чем другие отделы.

На совместной встрече QA и разработчиков обсудите следующие вопросы.

Автоматизированные интеграционные тесты.

Хотя за них обычно отвечает QA, разработчики также могут писать такие тесты, так как они хранятся в репозитории проекта. Важно разграничивать интеграционные (integration tests) и модульные тесты (unit tests): модульные тесты всегда должны писаться разработчиками.

Детализация шагов воспроизведения.

Нужно найти баланс между недостаточной и избыточной детализацией. Согласуйте оптимальный уровень с разработчиками и QA-инженерами, чтобы все понимали ожидания. Хотя в итоге вы совместно с командой QA будете углубляться в проблему, определение начальной точки упростит процесс.

Включение QA в основные встречи.

Участие членов команды QA в планировании (например, их присутствие на встречах по уточнению бэклога (backlog grooming) и ежедневных митингах) гарантирует их осведомленность о сроках, приоритетах и релизах. Они смогут оперативно сообщать о проблемах. Выносите обновления для QA в начало встреч (например, тикеты для тестирования) для оптимизации времени.

Важно, чтобы QA-специалисты чувствовали себя комфортно, обращаясь к разработчику, работавшему над тикетом, который они тестируют. Разработчики должны быть готовы совместно с командой QA определить, является проблема багом или чем-то другим (например, различием в окружении).

QA-инженер — лучший друг команды, так как он находит множество багов, которые могли бы вызвать проблемы в продакшене. Проблемы в продакшене никому не нужны. QA существует не для того, чтобы указывать разработчикам на ошибки или критиковать их код. Цель тестировщиков — убедиться, что нам не придется применять экстренные меры для тушения пожаров в продакшене. Будьте терпеливы, когда они сообщают о дефектах, — они просто выполняют свою работу.

При развертывании изменений вам предстоит взаимодействовать с разными командами — их состав зависит от компании. Например, с отделом продаж, чтобы помочь менеджерам понять возможности продукта и сформировать реалистичные ожидания у клиентов.

Также существует команда интеграции или команда по работе с клиентами (customer success team). Они обычно работают с клиентами индивидуально, помогая им интегрировать продукт в их системы. Им крайне важно быть в курсе любых критических изменений или новых функциональных возможностей.

В некоторых компаниях есть отдельная команда безопасности, особенно если продукт относится к финтеху, здравоохранению, электронной коммерции или производству. Эта команда обычно подключается сразу после QA для проведения security-тестов, а также может выполнять постоянное сканирование безопасности в рамках пайплайна разработки.

Служба поддержки (support team) — это та команда, которая так или иначе присутствует в любой организации. С ней вам нужно будет взаимодействовать на следующем и, вероятно, последнем этапе.

Планирование совместно со службой поддержки

Служба поддержки — это сотрудники, которые напрямую взаимодействуют с пользователями, столкнувшимися с проблемами. Часть сообщений о багах, которые вам предстоит исправить, поступает именно от них. Они могут даже привязать тикет поддержки к вашему баг-тикету, чтобы информировать пользователей о ходе решения. Кроме того, служба поддержки служит источником идей для новых функций. Если от разных пользователей постоянно поступают одинаковые запросы, стоит проработать решение.

Работа службы поддержки сложна — ее сотрудникам приходится взаимодействовать с недовольными клиентами. Ваша задача — минимизировать такие ситуации. При обсуждении грядущего релиза функции со службой поддержки важно ей объяснить, как это повлияет на пользователей. Продуктовая команда должна заранее согласовать с ней сроки. Ваша цель — выяснить, какая помощь требуется от команды разработки.

Помните: у сотрудников службы поддержки редко есть глубокие технические знания. Они разбираются в работе приложения с точки зрения пользователя и администратора, но не погружаются в технические детали. Понимание их уровня значительно упростит коммуникацию. Ваша задача — переводить технические аспекты на язык, который им понятен.

Вот несколько полезных рекомендаций.

Подготовьте документацию, которую служба поддержки сможет использовать при работе с функционалом.

Четко опишите все изменения по сравнению с текущей функциональностью. Особо выделите значения, которые потребуется вводить пользователям, поскольку именно пользовательский ввод часто становится причиной обращений в поддержку.

Внимательно выслушивайте жалобы, которые озвучивает служба поддержки.

Анализируйте поступающие от нее отчеты об ошибках и запросы, совместно с продакт-менеджером определяя приоритеты. Это не означает, что нужно реализовывать все пожелания поддержки, но имеет смысл исследовать причины запросов и оценить сложность исправлений.

Всегда согласовывайте даты и время релизов.

Служба поддержки часто выделяет дополнительное время перед релизами, ожидая увеличения числа вопросов. Им необходимо точно знать сроки выхода продукта, чтобы запланировать это время.

Проводите демонстрацию готового функционала непосредственно перед выпуском.

После утверждения изменений для релиза организуйте краткую живую демонстрацию для поддержки (и остальных заинтересованных сторон). Это позволит им увидеть корректную работу приложения и задать уточняющие вопросы.

Выделите дополнительное время разработчиков для помощи службе поддержки в дни релизов.

Если нагрузка на поддержку станет критической, будьте готовы откатить изменения — внимательно отслеживайте ситуацию.

Обсуждение этих вопросов со службой поддержки — еще один способ выявить проблемы на ранних этапах проекта. Кроме того, это отличная возможность укрепить взаимодействие между командами разработки и поддержки, поскольку на практике вы сотрудничаете теснее, чем может показаться. Это последняя команда, с которой вам нужно согласовать действия. Теперь пришло время систематизировать все полученные данные, чтобы составить целостный план.

Заключение

Итак, финальный этап запуска проекта позади. Теперь, имея заметки после встреч с разными командами, нужно обобщить информацию и выделить ключевые моменты для продакт-менеджеров.

Все прошедшие обсуждения помогут определить реалистичные сроки. На этом этапе начнутся переговоры. Важно помнить: в разработке всегда что-то идет не по плану. Заложите временной буфер для себя и команды — это позволит спокойно работать, а возможно, даже закончить процесс раньше обещанного срока. Принцип «обещай меньше, делай больше» остается актуальным. Если только функция не привязана к жесткому дедлайну (например, обновление стороннего сервиса), добавьте несколько дней на непредвиденные обстоятельства. После согласования сроков с продакт-менеджерами (с учетом мнений всех команд) соберите участников на итоговую короткую встречу.

Важно понимать, что установка сроков в разработке ПО — крайне сложная задача. В любой момент может возникнуть множество непредвиденных обстоятельств, которые сорвут все планы. Если кто-то заболеет или из-за стихийного бедствия у сотрудника на неделю пропадет электричество, часть тикетов останутся невыполненными. В процессе разработки может выясниться, что интеграция с определенным инструментом гораздо сложнее, чем предполагалось. Или же другая команда разработчиков задержит свою часть работы, что автоматически сдвигает и ваши сроки.

И это нормально. Главное — как можно быстрее сообщить о возникших проблемах всем заинтересованным сторонам. После этого можно начинать планировать дальнейшие действия. Жизнь есть жизнь, и нужно стараться не позволять таким ситуациям выбивать вас из колеи каждый раз, потому что они будут происходить постоянно!

В этой встрече должны участвовать представители всех команд, с которыми вы взаимодействовали. На данном этапе нужно лишь подтвердить назначенные задачи и сроки для каждого. Основная цель — закрепить ответственность и трижды убедиться, что все участники одинаково понимают ситуацию. В качестве подготовки к встрече соберите всю информацию из своих заметок и создайте краткий документ, которым все смогут пользоваться вплоть до релиза.

К этому моменту у вас уже есть множество тикетов для работы, причем четко определенных. Вы задокументировали все межкомандные взаимодействия. Фаза старта проекта завершена. Теперь у вас есть все необходимое, чтобы наконец приступить к написанию кода. В следующей части книги вы развернете масштабируемый бэкенд, используя фреймворк NestJS и язык TypeScript.

Часть II

Разработка бэкенда

ГЛАВА 2

Настройка бэкенда

https://t.me/it_boooks/2

Предположим, у вас уже готовы все тикеты и вам поручили создать «скелет» бэкенда. В вашей компании используют фреймворк NestJS (https://nestjs.com). Не переживайте, если вы никогда не работали с NestJS или даже не слышали о нем, — документация качественная, и у вас будет доступный для изучения код. Обратитесь к другим разработчикам, чтобы понять, почему в существующих бэкендах были приняты те или иные архитектурные и дизайнерские решения.

В этой главе вы узнаете:

- как принимать архитектурные решения для бэкенда;
- как проверить работоспособность приложения после первоначальной настройки;
- как писать начальную документацию.

Эти задачи критически важны для обеспечения бесперебойной разработки, по мере того как и продукт, и команда будут расти и меняться. Все, что вы будете создавать в бэкенде, будет опираться на фундамент, заложенный здесь, поэтому не спешите и детально обсудите варианты с командой.

Почему NestJS?

Прежде чем углубляться в детали, давайте обсудим, почему NestJS — это надежный выбор. По сути, это готовая бэкенд-архитектура «из коробки». Сразу после инициализации приложения вы получаете доступ к валидации, аутентификации, маршрутизации, контроллерам, схеме данных и множеству других функций. Вместо того чтобы собирать все вручную, NestJS предоставляет все необходимое для создания масштабируемого бэкенда с самого начала. Это означает чуть меньшую гибкость в организации кода, но это компромисс за готовую инфраструктуру. Именно здесь начинается интересная часть — настройка исходного репозитория для бэкенда.

Принимаемые на этом этапе решения определяют будущее проекта. При выборе опций запрашивайте профессиональное мнение команды. Коллеги могут знать то, чего не знаете вы. Это не только улучшит приложение, но и поможет команде учиться.

В разработке бэкенда так много нюансов, что альтернативные мнения значительно облегчат вашу работу. Также полезно вести записи архитектурных решений (ADR, architecture decision records), в которых фиксируются дата и причины принятия наиболее важных. Примеры можно найти в GitHub-репозитории (https://oreil.ly/C78GT).

Создание нового проекта с чистого листа может вызывать чувство некоторой растерянности. Перед вами лишь пустой терминал, а опереться можно только на примеры и шаблонный код. Направление развития проекта полностью зависит от решений вашей команды. Однако это захватывающий этап — вы получаете возможность создать чистую, удобную для обновлений, тестирования и масштабирования кодовую базу.

Выбор архитектурного подхода

В этом проекте используется монолитная бэкенд-архитектура с несколькими serverless-функциями. Такой гибридный подход ускоряет разработку и развертывание, позволяя одновременно освоить разные модели. Альтернативой мог бы стать микросервисный подход, где все API реализуются через serverless-функции. Выбор архитектуры всегда определяется долгосрочными планами развития приложения.

Микросервисы требуют больше ресурсов и разработчиков для поддержки по сравнению с монолитами, но идеально подходят для проектно-ориентированной архитектуры (глава 30). Монолиты усложняют внесение масштабных изменений — в этом они проигрывают микросервисной модели.

Для разработки фулстек-приложения мы будем использовать TypeScript (http://typescriptlang.org). TypeScript выбран в качестве языка программирования, поскольку он упрощает отладку за счет снижения вероятности ошибок — система типов не позволит скомпилировать некорректный код. Строгая типизация помогает выявлять ошибки еще до запуска программы. Кроме того, TypeScript наглядно отображает поля, типы данных и их источники, что критически важно для поддержания согласованности типов между фронтендом и бэкендом. Встроенная система типов работает как документация, показывая связи между компонентами и доступные параметры.

Настройка NestJS

Мы детально разберем все этапы инициализации проекта. Для начала необходимо подготовить окружение. Вы можете работать с демонстрационным репозиторием (https://oreil.ly/-Czox). Первым шагом установите NestJS CLI. На момент написания книги актуальная версия фреймворка — 9.2.0. Выполните в терминале следующую команду:

```
npm i -g @nestjs/cli
```

Новый проект NestJS создается командой:

```
nest new <ваше-название-проекта>.
```

В нашем случае выполним:

```
nest new dashboard-server
```

NestJS запросит выбор менеджера пакетов. Укажите `npm` и нажмите Enter:

```
? Which package manager would you 🖤 to use? (Use arrow keys)
› npm
  yarn
  pnpm
```

Отслеживание версий пакетов — это то, что senior-разработчик делает по умолчанию. В процессе создания приложения вы познакомитесь с наилучшими практиками разработчика, такими как поддержание документации в актуальном состоянии и контроль корректности локального окружения у всех участников. Также вам потребуется настроить линтинг с помощью ESLint (https://eslint.org), единые правила форматирования через инструменты вроде Prettier (https://prettier.io) для репозитория, а также различные конфигурации окружения для обеспечения согласованности кодовой базы.

После этого начнется установка, которая может занять несколько минут. Этот процесс включает установку всех зависимостей проекта, создание каркаса архитектуры, добавление шаблонного кода и другие операции. Система даже предоставляет предварительно настроенный TypeScript. Прежде чем вносить изменения, уделите время изучению репозитория. Независимо от того, работаете вы с новым приложением, как в нашем случае, или с legacy-проектом, всегда начинайте с анализа общей структуры.

Основные моменты для анализа:

- файл `package.json` — чтобы понять, какие пакеты используются в проекте;
- все конфигурационные файлы — для изучения внутренней логики работы;
- тестовые файлы — они дают четкое представление о ключевой функциональности приложения;
- непосредственно код — чтобы оценить принятые стандарты разработки и возможные улучшения.

После предварительного изучения проекта можно запустить приложение следующими командами:

```
cd dashboard-server
npm start
```

В терминале появится примерно такой вывод, подтверждающий успешный запуск приложения:

```
> dashboard-server@0.0.1 start
> nest start

[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [NestFactory] Starting Nest
application...
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [InstanceLoader] AppModule
dependencies initialized +11ms
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [RoutesResolver] AppController {/}:
+5ms
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [RouterExplorer] Mapped {/, GET}
route +3ms
[Nest] 20891  - 03/03/2023, 8:18:46 PM     LOG [NestApplication] Nest application
successfully started +1ms
```

Здесь возникает первый важный архитектурный вопрос: как именно вы будете тестировать этот бэкенд? Согласованное окружение разработки напрямую влияет на скорость тестирования кода и снижает уровень фрустрации команды по мере роста как продукта, так и самой команды.

Стандартный выбор — использовать Docker (https://oreil.ly/iMG1T), потому что вы можете настроить все: от базы данных Postgres до самого веб-приложения с бэкендом — в одном контейнере (или в нескольких). Эти контейнеры могут содержать абсолютно одинаковые данные и версии и разворачиваться в любом локальном окружении, независимо от операционной системы. Другой вариант — использовать функционал VS Code под названием Dev Containers (https://oreil.ly/c4Bck). В зависимости от команды и проекта это тоже может быть хорошим выбором.

Локальное тестирование бэкенда

Среди популярных инструментов тестирования — Postman (https://oreil.ly/17Yn0) и RapidAPI (https://paw.cloud). В этой книге (как и во многих реальных проектах) мы будем использовать Postman, так как он имеет бесплатную версию. С его помощью вы сможете отправлять запросы к своим эндпоинтам с разными заголовками, телами и другими параметрами, после чего анализировать получаемые ответы. Именно так обычно тестируют бэкенд до появления готового пользовательского интерфейса.

Помните: такие инструменты нужны для ускорения работы. Не зацикливайтесь на выборе конкретного решения, главное, чтобы локальная разработка была эффективной. Теперь, если вы настроили Postman, отправьте GET-запрос на http://localhost:3000. В ответе должно появиться `"Hello World!"`, как показано на рис. 2.1.

У каждого разработчика свои предпочтения в инструментах для тестирования бэкенда, и это нормально. Однако согласованность внутри команды упрощает диагностику проблем: вы можете обмениваться тестами эндпоинтов, что ускоряет и стандартизирует поиск ошибок. В дальнейшем эти тесты станут частью документации для фронтенда. Такую задачу стоит включить в спринт через отдельный тикет.

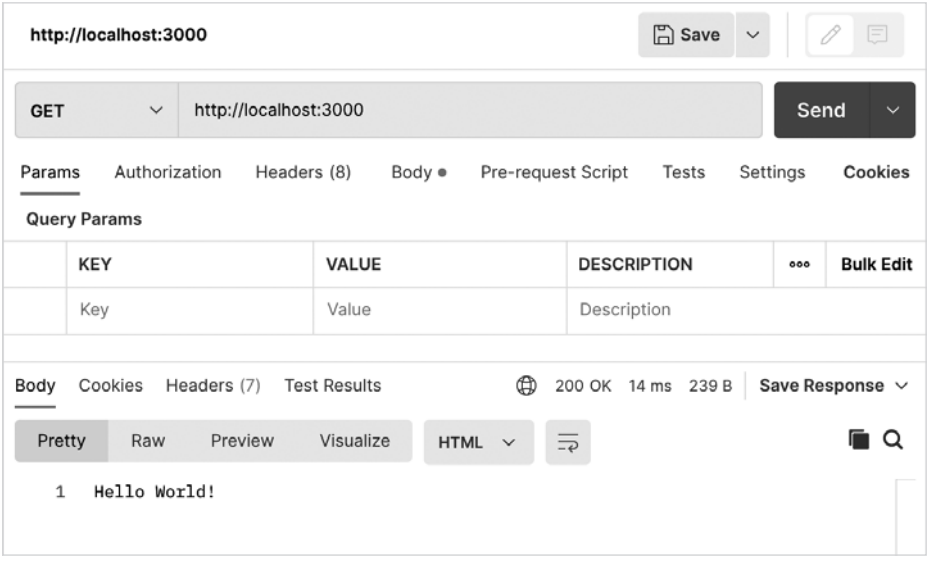

Рис. 2.1. GET-запрос и ответ в Postman

Теперь, когда работоспособность приложения подтверждена, можно дорабатывать его для команды.

Обновление README

Начните с обновления файла `README.md`, добавив конкретные инструкции по настройке и запуску приложения. В нем уже есть много полезной информации, поэтому немного сократите текст и добавьте несколько новых пунктов. Информация о базовом обновлении может выглядеть так:

```
# Dashboard Server

## Описание

Бэкенд для поддержки клиентов, созданный на основе стартового репозитория
TypeScript фреймворка Nest.

## Установка

```bash
$ npm install
```

Запуск приложения

```bash
# режим разработки
```
```

```
$ npm run start

# режим отслеживания изменений
$ npm run start:dev

# продакшен-режим
$ npm run start:prod
```

Тестирование

```bash
# модульные тесты
$ npm run test

# end-to-end тесты
$ npm run test:e2e

# проверка покрытия тестами
$ npm run test:cov
```

Это не должен быть объемный документ с философскими размышлениями о коде или проекте. Достаточно дать четкие инструкции, чтобы любой разработчик мог клонировать репозиторий и запустить приложение. Остальная документация будет храниться отдельно. По мере роста репозитория в README можно добавлять детали — это живой документ, который разрешено обновлять через пул-реквесты (pull requests) при необходимости.

Добавление CHANGELOG

Мы рекомендуем включать файл `CHANGELOG.md` в ваш репозиторий. Он будет хранить детальную запись всех изменений в каждом релизе. Наличие такой записи поможет определить, какие обновления содержатся в каждой версии приложения, что особенно полезно при возникновении проблем в продакшене. Вот как может выглядеть начало этого файла:

```
# CHANGELOG

## Руководство

- Мажорные релизы содержат критические изменения, и номер версии увеличивается до
`x.0.0`
- Минорные релизы добавляют новые функции без критических изменений, и номер версии
увеличивается до `0.x.0`
- Патч-релизы включают исправления ошибок и улучшения производительности, и номер
версии увеличивается до `0.0.x`

### 0.0.1

- Первоначальный релиз
```

Также стоит рассмотреть установку конкретных правил для сообщений о коммитах. Если вам нужен определенный формат, настройте инструмент вроде commitlint (https://commitlint.js.org/#/). Это позволит включать в сообщения детали: например, указывать, является ли коммит исправлением (fix) или технической задачей (chore). При необходимости можно добавлять номера тикетов. Вы также можете настроить требования, например, чтобы сообщение начиналось с глагола, описывающего суть изменения.

Сообщение о коммите может выглядеть так:

```
fix: обновление модального окна для однократной отправки API-запроса
```

Возможно, вам придется отвечать за подготовку пул-реквестов для релизов, поэтому важно убедиться, что обновление этого файла включено в процесс. Хороший способ не забывать об этом — обновлять `CHANGELOG.md` одновременно с версией приложения в `package.json`.

Во многих проектах потребуется настройка конфигурационных файлов ESLint, Prettier и TypeScript. В этом приложении уже предусмотрены стандартные настройки, но их стоит проверить и при необходимости добавить или удалить параметры. На этом первоначальная настройка приложения завершена, и можно переходить к проектированию схемы данных.

Монолитная и микросервисная архитектуры

Это ключевой этап проекта, так как вы закладываете основу всей дальнейшей разработки. Необходимо продумать перспективы развития проекта, а также потенциальные ограничения, которые могут возникнуть при масштабировании.

Выделите время на создание архитектурной диаграммы. Она не обязана быть сложной, но должна отображать взаимодействие бэкенда со всеми компонентами системы: от фронтенда до cron-задач. *Cron-задачи* — это фоновые процессы, выполняющиеся по расписанию. Их также можно реализовать на TypeScript.

Бэкенд — это своего рода центр управления. Он определяет развитие всех остальных компонентов, поскольку взаимодействует со множеством частей системы. Именно поэтому так важно не торопиться и тщательно обдумать этот выбор. Две наиболее распространенные архитектуры, с которыми вы столкнетесь, — это монолитная и микросервисная. Далее я расскажу об их различиях. Рассматривайте это как руководство для выбора подходящего варианта.

Монолитная архитектура (monolith) предполагает размещение всей бэкенд-логики в единой кодовой базе. Такой подход часто встречается в корпоративных приложениях. Здесь используется одна база данных и один сервер для всего приложения. Развертывание происходит как единый блок, подключенный ко всем используемым сервисам. В нем обрабатываются все запросы к эндпоинтам, фоновые задачи (cron jobs) и интеграции со сторонними сервисами.

Главное преимущество монолита — вся кодовая база находится в одном месте. Это позволяет понимать, как работает система в целом и где требуются изменения. Инфраструктура настраивается проще, так как развертывается только одно приложение. Отладка также упрощается благодаря возможности анализировать бэкенд как единое целое.

Создание монолита — более простой подход, но у него есть недостатки. Например, если баг попадает в продакшен и во время выполнения вызывает ошибки, которых не было локально, все приложение становится недоступным для пользователей. Также монолит сложно масштабировать. Возможна ситуация, когда лишь один эндпоинт получает высокую нагрузку, но при монолитной архитектуре приходится масштабировать весь бэкенд, что может быть дорого. На рис. 2.2 показан пример монолитного бэкенда.

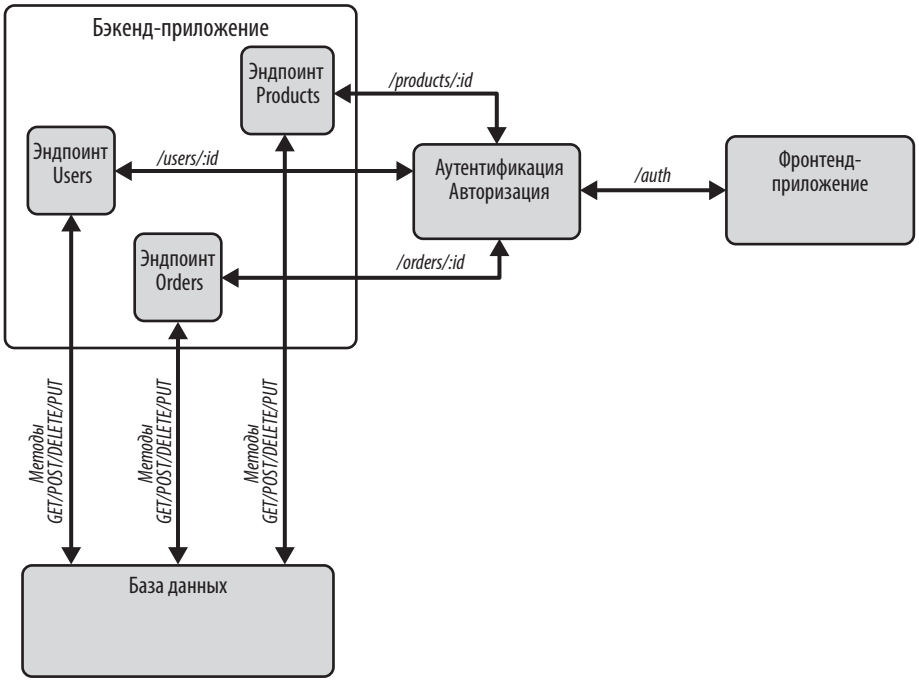

Рис. 2.2. Архитектурная схема простого монолита

Микросервис (microservice) — это полная противоположность монолиту. Если в монолите есть директория или набор файлов, отвечающих за определенную бизнес-логику, то в микросервисной архитектуре они становятся самостоятельным API. В микросервисах вся бэкенд-логика разделена на отдельные функциональные блоки. Каждый микросервис имеет собственную кодовую базу, базу данных и сервер.

Одно из главных преимуществ микросервисов — масштабируемость. Поскольку бизнес-логика разделена на отдельные сущности, можно увеличивать ресурсы для одного эндпоинта, не затрагивая другие. Уникальная возможность — использовать разные языки программирования для разных микросервисов. Например, микросервис для обработки большого количества запросов может быть написан на Rust, микросервис для машинного обучения — на Python, а основной микросервис — на TypeScript, если это основной язык команды.

Микросервисы обеспечивают гибкость на всех этапах разработки и развертывания. Однако у них есть и недостатки. Отладка микросервисов может быть сложной, поскольку поиск источника проблемы требует анализа всех работающих микросервисов. Также могут возникнуть проблемы с целостностью данных, если одни и те же данные используются в нескольких микросервисах. На рис. 2.3 показан пример архитектуры бэкенда на основе микросервисов.

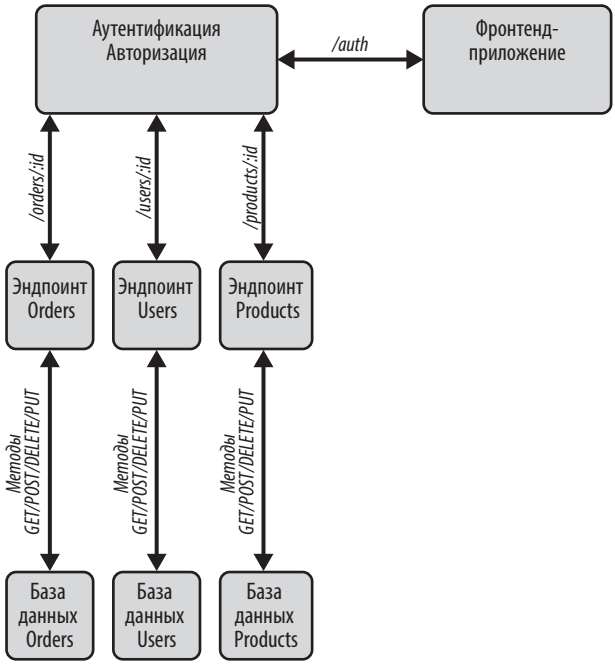

Рис. 2.3. Архитектурная схема простых микросервисов

Альтернативные архитектуры для рассмотрения

Еще одна архитектура, о которой вы услышите, — это *serverless-архитектура*. Ее можно использовать для всей бэкенд-функциональности, но важно учитывать различия, так как это существенно влияет на затраты у облачного провайдера. Например, несколько lambda-функций для специфичных задач могут удачно

дополнить ваш бэкенд. По мере роста продукта serverless-архитектуру можно успешно сочетать с монолитной.

До появления микросервисов существовала *сервис-ориентированная архитектура* (*SOA*). Ее ключевое отличие — все сервисы используют общую базу данных. Это может быть полезно для сложных приложений, где разделение функциональности на небольшие модули упрощает поддержку. Мы не будем подробно разбирать SOA (см. https://oreil.ly/lYFoW) — вы можете самостоятельно оценить, насколько она применима для вашей архитектуры.

Архитектура хороша тем, что вы можете комбинировать именно те элементы, которые подходят для вашего кейса. Работая с микросервисами, вы также можете столкнуться с концепцией сервисных сеток (service meshes). Рекомендую изучить их детальнее (https://oreil.ly/e5cGC), но в этой книге они рассматриваться не будут.

Помните: проектирование бэкенда — это длительный процесс. Его не стоит торопить, и крайне важно собирать обратную связь от других разработчиков и команд. Выбранная архитектура определяет будущее развитие проекта, так как последующая смена всей системы — сложная задача.

Выбор шаблона проектирования API: REST и GraphQL

На этом этапе также необходимо выбрать способ обмена данными: REST API, GraphQL или другой подход. Это решение повлияет на взаимодействие фронтенда и сервисов с бэкендом, а также на масштабируемость кода при добавлении новых ролей, прав доступа и функционала.

Сегодня REST является стандартом в бэкенд-разработке, а GraphQL прочно занял второе место. Существуют и другие варианты, например SOAP API (Simple Object Access Protocol), но, надеюсь, вам не придется с ними работать.

Ключевое отличие REST от GraphQL заключается в том, что GraphQL использует единый эндпоинт для всех запросов, тогда как REST — множество отдельных эндпоинтов. Хотя эти парадигмы часто противопоставляют, на практике разница сводится к способам построения и применения API. Важно отметить, что GraphQL изначально поддерживает подписки (subscriptions), тогда как в REST для аналогичной функциональности потребуются WebSockets (https://oreil.ly/_vq7K) или схожие технологии.

В REST API каждый эндпоинт обычно соответствует одному типу ресурса. Например, эндпоинт `/orders` в вашем проекте обрабатывает GET, POST и PUT запросы для взаимодействия с данными заказов. В GraphQL эти эндпоинты преобразуются в схему следующего вида:

```
type Order {
  id: ID
  name: String
  total: Number
```

```
  products: Product[]
  createdAt: Date
  updatedAt: Date
}

type CreateOrderInput {
  name: String
  total: Number
  products: Product[]
}

type UpdateOrderInput {
  name: String
  total: Number
  products: Product[]
}

type Query {
  order(id: ID!): Order
  orders: Order[]
}

type Mutation {
    create(input: CreateOrderInput): Order
    update(input: UpdateOrderInput): Order
}
```

В GraphQL вы указываете нужную операцию с помощью ключевых слов `query` или `mutation` вместо HTTP-методов. Получаемые данные при этом не отличаются от HTTP-запросов — ответ GraphQL по-прежнему будет в формате JSON с теми же значениями, что и HTTP-ответ. Разница заключается в способе управления кодом.

При работе с REST вы указываете путь к эндпоинту, метод HTTP и передаваемые данные. GraphQL предоставляет бóльшую гибкость: вместо путей к эндпоинтам вы пишете функции-резолверы для запросов (queries) и изменений (mutations). Вот пример резолверов для работы с заказами:

```
{
  Query: {
    order: (root, { id }) => find(orders, { id: id }),
  },
  Order: {
    products: (order) => filter(products, { orderId: order.id }),
  },
}
```

Соответствующий запрос будет выглядеть так:

```
query {
  order(id: "fejiw-f4wt301-4tfw2g-g4t24") {
    name
    products {
```

```
      name
      price
    }
  }
}
```

В этом и заключается часть силы GraphQL. Вы можете запрашивать именно те данные, которые вам нужны, без получения лишних значений. В REST-запросах нельзя указать, какие именно одно-два значения должны вернуться, — каждый раз вы получаете все данные с эндпоинта. GraphQL позволяет сократить время отклика в системах, обрабатывающих огромные объемы взаимосвязанных данных, поскольку запрашивается только необходимое.

Эта книга не будет подробно рассматривать GraphQL: мы просто обозначим, что такая технология существует, и в IT-индустрии она широко распространена. Можно считать GraphQL следующим этапом эволюции после REST — аналогично тому, как TypeScript стал стандартом вслед за JavaScript. Стоит потратить время на его изучение, поэтому ознакомьтесь с ресурсами GraphQL на официальном сайте (https://graphql.org), где вы найдете сообщества, курсы, инструменты и документацию.

Заключение

В этой главе мы познакомились с приложением NestJS, которое вы будете настраивать для своего бэкенда. Мы рассмотрели различные архитектуры и парадигмы API, а также связанные с ними компромиссы. Теперь, когда выполнена первоначальная настройка бэкенда NestJS и понятны причины каждого решения, пришло время перейти к схеме данных.

ГЛАВА 3

Разработка схемы данных

Слой данных охватывает множество аспектов, и хорошо, что у нас есть столько отличных инструментов для работы. Независимо от выбранных инструментов, важно понимать, что происходит «под капотом». При проектировании схемы данных не спешите — тщательно продумывайте ее. Схема данных определяет работу всего приложения и его зависимостей. Чем дальше, тем сложнее будет ее менять без ущерба для функциональности.

Поскольку вы уже определились с архитектурой бэкенда, у вас, вероятно, есть представление о взаимосвязях данных и их структуре. Создание диаграммы этих связей значительно упростит работу вам и команде, так как наглядно покажет отношения между сущностями.

В этой главе мы рассмотрим:

- первоначальные аспекты проектирования схемы данных;
- настройку базы данных;
- использование ORM-инструментов (Object-Relational Mapping);
- создание миграций БД;
- заполнение БД начальными данными (сидирование).

Стремитесь к максимальной детализации и документированию бизнес-логики. Регулярно запрашивайте обратную связь от команды — схема может усложняться по мере роста приложения. Убедитесь, что все понимают, как, где и почему хранятся данные. К концу главы у вас будет надежный фундамент для построения схемы данных.

С чего начать

Для разработки слоя данных выполните следующие основные шаги.

1. Создайте диаграмму схемы данных. Она должна включать сущности, их столбцы с типами данных и взаимосвязи.
2. Настройте подключение к базе данных в приложении. В примере используется Prisma (https://oreil.ly/i7lYw) для PostgreSQL (https://postgresql.org), но есть и другие инструменты: Knex.js (https://knexjs.org) или Drizzle (https://oreil.ly/5ye9-).

PostgreSQL выбран за открытый исходный код и многолетнюю надежность — его используют как в больших, старых системах, успешно работающих не годы, а десятилетия, так и в новых приложениях.

3. Реализуйте схему данных в коде, преобразовав получившуюся диаграмму.
4. Добавьте начальные данные (seed data). Это обеспечит наличие критически важных данных с самого начала и упростит тестирование в разных средах.
5. Запустите миграции после настройки подключения, подготовки схемы и начальных данных.
6. Протестируйте базу простыми SQL-запросами. Проверьте создание/обновление таблиц, особое внимание уделите корректности связей через первичные (primary) и внешние (foreign) ключи.

Некоторые приложения будут иметь более сложный сценарий, но во всех проектах вы встретите схожий процесс. Благодаря обсуждениям с продуктовой командой вы уже знаете, какие данные ожидаете, поэтому сейчас самое время создать качественную диаграмму для команды разработчиков (dev team).

В этой книге мы не будем рассматривать нереляционные базы данных (nonrelational databases), так как работаем с реляционными (relational databases). Реляционные базы накладывают строгие правила на взаимосвязи данных, в отличие от нереляционных. Выбор между ними зависит от типа проекта. Если вам нужно хранить данные произвольного формата (например, события с постоянно меняющейся информацией), нереляционные базы обеспечивают бóльшую гибкость. У нереляционных баз есть специфические сценарии использования, но большинству приложений достаточно стандартной реляционной базы.

Разработка схемы данных

Диаграмма служит отличным визуальным ориентиром для разработчиков и QA, помогая понять, какие значения ожидать и почему. Это эффективный инструмент для обсуждений между фронтенд- и бэкенд-разработчиками, а также для документирования связей данных без использования кода. Повторим: диаграммы не должны быть излишне сложными. Разработчики часто зацикливаются на незначительных деталях (например, инструментах для создания диаграмм или форматах изображений). Важно вовремя отследить, если вы углубились в задачу, не требующую столь пристального внимания.

Ваша диаграмма может быть простой, если она включает таблицы, столбцы, типы данных и связи между таблицами. По возможности используйте инструменты с прямым подключением к базе данных для визуализации, например DBeaver (https://oreil.ly/GRPz9). Так вы увидите точные связи, которые определили. В качестве альтернативы можно продолжить использовать Miro для хранения всей архитектурной документации в одном месте. Главное — чтобы формат был понятен всем.

На рис. 3.1–3.3 показаны примеры документации для разных таблиц и их взаимосвязей:

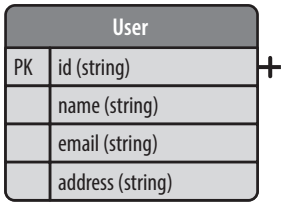

Рис. 3.1. Столбцы таблицы пользователей (User)

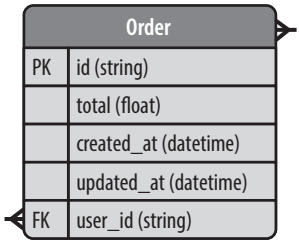

Рис. 3.2. Столбцы таблицы заказов (Order)

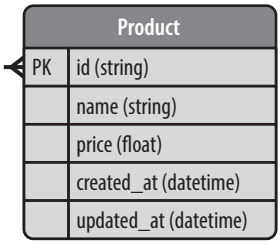

Рис. 3.3. Столбцы таблицы товара (Product)

На рис. 3.4 отображены связи между всеми таблицами: User, Order и Product.

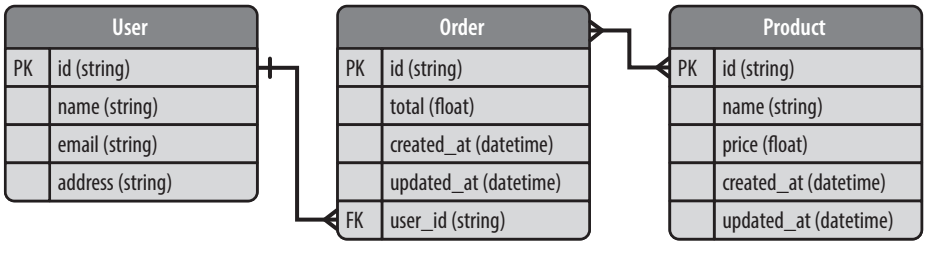

Рис. 3.4. Связи между всеми таблицами: User, Order и Product

После документирования схемы данных обсудите ее с командой: изложите свои идеи и запросите обратную связь. Фронтенд-разработчики могут захотеть получить специфические форматы данных для рендеринга элементов. Хотя схема не должна проектироваться исключительно под фронтенд, важно учитывать в ответах эндпоинтов запросы фронтенда (поиск, сортировку). Другие разработчики могут поднять вопросы безопасности.

Совместное обсуждение создает более продуманную схему, чем индивидуальная работа. Вы научитесь увереннее делиться идеями и вовлекать коллег. Не бойтесь ошибаться во время обсуждений — ваши ошибки могут вдохновить других на новые решения.

Первое время это может вызывать дискомфорт — кажется, будто все пристально оценивают ваш код и навыки, хотя на самом деле это не так. Важно привыкнуть к конструктивной обратной связи: именно она помогает вам и команде создавать более качественный код. Чем чаще вы делитесь своими решениями и идеями с командой, тем быстрее сможете улучшить рабочие процессы для всех.

После согласования схемы данных между фронтенд- и бэкенд-разработчиками можно приступать к подключению бэкенда к базе данных для создания таблиц. Вы будете работать с Postgres у себя локально, но при необходимости его можно развернуть и в облаке.

Настройка Postgres

Настройка Postgres необходима для получения данных подключения (connection information) вашего приложения. Обычно этим занимается команда, отвечающая за инфраструктуру продакшен и других сред. Однако локальный экземпляр нужен для проверки корректности изменений, вносимых вами и командой. Для согласованности локальных окружений можно использовать Docker-контейнер (https://oreil.ly/beu9J). Также важно продумать процесс адаптации новых разработчиков: настройка локального окружения — одно из основных препятствий при вхождении в команду, поэтому инвестиции в это окупятся по мере роста продукта и команды.

Postgres можно скачать бесплатно (https://oreil.ly/qZocC). Следуйте инструкциям в документации, чтобы все настроить, запустить и задать мастер-пароль (master password). Базовая настройка безопасности локальной БД помогает выявить потенциальные проблемы на раннем этапе, так как вы сразу учитываете требования продакшен-среды. После установки pgAdmin откройте приложение для создания новой базы данных. Затем раскройте выпадающее меню PostgreSQL 14, кликните правой кнопкой на раздел Databases и выберите создание новой БД с именем dashboard. Результат будет аналогичен показанному на рис. 3.5.

После создания новой базы данных dashboard вам потребуется запомнить ее имя, пароль, номер порта и имя пользователя для настройки подключения к бэкенду.

Если вы не изменяли значения по умолчанию, часть учетных данных будет следующей:

- имя пользователя БД: postgres;
- номер порта БД: 5432.

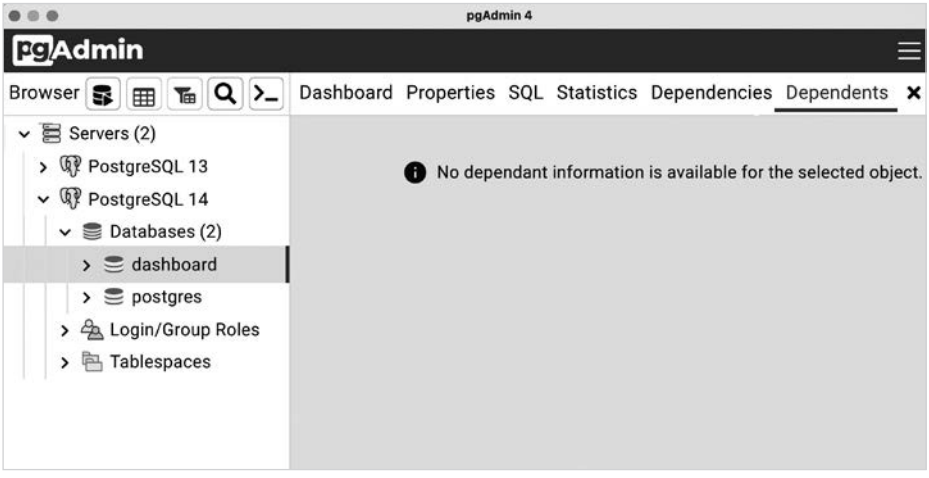

Рис. 3.5. База данных dashboard в локальном Postgres

Теперь можно приступать к созданию таблиц, схема которых была подготовлена ранее. Далее следует использовать инструмент ORM для подключения бэкенда к базе данных и проверки корректности настройки.

Базовые SQL-запросы

Важно знать основные SQL-команды. Углубленного изучения таких концепций, как представления (views) и индексы, не потребуется, базовых навыков выполнения операций CRUD (создание, чтение, обновление, удаление) будет достаточно для большинства задач. Начнем с написания запроса на добавление новой строки в таблицу. Для этого используется следующий SQL-запрос:

```
INSERT INTO table_name (column1, column2, column3) VALUES (value1, value2,
value3);
```

Замените *`table_name`* на имя целевой таблицы. Поля *`column1`*, *`column2`* и *`column3`* соответствуют названиям столбцов, а *`value1`*, *`value2`* и *`value3`* — значениям для вставки. Для тестирования можно добавить строку следующим образом:

```
INSERT INTO Orders (id, name, total)
VALUES (4, 'Mark', 25.99)
RETURNING id;
```

Для проверки корректности сохранения данных выполните запрос:

```
SELECT * FROM Orders;
```

Для завершения работы с примером может потребоваться удаление тестовых данных. Используйте запрос вида:

```
DELETE FROM Orders
WHERE id = 4
RETURNING *;
```

Уверенное владение такими командами позволяет проверять данные напрямую в БД. Если вы хотите вникнуть в работу SQL, используйте ресурсы вроде SQLBolt (https://sqlbolt.com) или LearnSQL.com, но это не обязательно.

Выбор ORM

Помимо выбора фреймворка для бэкенда, выбор инструмента ORM (Object-Relational Mapping) — одно из ключевых решений для будущего вашего проекта. Переход на другой ORM после начала разработки — нетривиальная задача. Если у вас нет особых предпочтений, то можно использовать NestJS, который имеет встроенную поддержку TypeORM (https://oreil.ly/LdFeM), Sequelize (https://sequelize.org) и Mongoose (https://mongoosejs.com/docs). Среди других популярных ORM-инструментов — Knex.js (https://knexjs.org) и Prisma (https://prisma.io).

Все эти инструменты выполняют схожие функции, но с разными подходами. Выбор зависит от опыта команды, удобства использования и ограничений инструментов. Для нашего проекта выбрана Prisma: команда уже имеет опыт работы с ней, документация поддерживается на высоком уровне, кроме того, Prisma применяют разработчики в различных сферах. Хотя NestJS имеет встроенный ORM, Prisma предлагает бóльшую поддержку, активное сообщество и отличную совместимость с Postgres. Эти преимущества определили наш выбор.

Для начала установите Prisma и TSX как dev-зависимости:

```
npm install prisma @prisma/client tsx --save-dev
```

Есть несколько важных нюансов, особенно касающихся зависимостей, о которых следует знать. При добавлении зависимости в проект убедитесь, что понимаете ее тип. Существует три типа зависимостей:

- основные зависимости (regular);
- зависимости для разработки (dev);
- парные зависимости (peer).

> *Основные зависимости* требуются приложению для работы в среде продакшен. *Зависимости для разработки* (например, инструменты тестирования или линтинга) не нужны в рабочей версии приложения. *Парные зависимости* становятся актуальными при публикации пакета — это зависимости, которые ваш пакет ожидает найти в родительском приложении.

После установки Prisma в проект требуется настроить подключение к PostgreSQL. Как и с любым ORM, необходимо выполнить инициализацию. Для Prisma используйте команду (пример вывода актуален для текущей версии):

```
npx prisma init --datasource-provider postgresql

✔ Your Prisma schema was created at prisma/schema.prisma
You can now open it in your favorite editor.

warn You already have a .gitignore file. Don't forget to add `.env` in it to not
commit any private information.

Next steps:
1. Set the DATABASE_URL in the .env file to point to your existing database. If
  your database has no tables yet, read https://pris.ly/d/getting-started
2. Run prisma db pull to turn your database schema into a Prisma schema.
3. Run prisma generate to generate the Prisma Client. You can then start
  querying your database.

More information in our documentation:
https://pris.ly/d/getting-started
```

Всегда читайте вывод консоли после выполнения команд. Он часто содержит полезные подсказки!

Эта команда создаст каталог prisma и файл `.env` в вашем проекте. Убедитесь, что `.env` добавлен в `.gitignore`. В каталоге prisma вы найдете `schema.prisma` — файл для настройки подключения к БД и моделей приложения. Файл `.env` содержит URL базы данных (строку подключения с учетными данными). Обновите значение `DATABASE_URL` в `.env` своими локальными данными, например:

```
DATABASE_URL="postgresql://username:password@localhost:5432/dashboard"
```

Все значения из `.env` должны обрабатываться в CI/CD-пайплайне. Согласуйте с командой DevOps настройку под вашу инфраструктуру. Вот пример для GitHub Actions (https://oreil.ly/R4d_u):

```
name: Node.js CI

on:
  push:
    branches: [ "main" ]
```

```
  pull_request:
    branches: [ "main" ]

env:
  DATABASE_URL: ${{ secrets.ProdDatabase }}

jobs:
  build:
    runs-on: ubuntu-latest
```

Теперь перейдите к файлу `schema.prisma` и начните описывать модели. Самый сложный этап (проектирование связей) уже завершен, поэтому можно приступать к созданию кода. Здесь проявляется ключевое отличие ORM-инструментов: у Prisma свой уникальный стиль, к которому нужно привыкнуть, и в этом вам поможет документация.

В файле `schema.prisma` вы можете начать добавлять части вашей модели. Добавьте следующий код в конец файла:

```
// schema.prisma
…
model User {
  id          Int        @id @default(autoincrement())
  email       String     @unique
  name        String
  address     String
  orders      Order[]
}

model Order {
  id              Int        @id @default(autoincrement())
  total           Float
  createdAt       DateTime   @default(now())
  updatedAt       DateTime   @updatedAt
  products        Product[]
  userId          Int?
  User            User?      @relation(fields: [userId], references: [id])
}

model Product {
  id              String     @id @default(uuid())
  name            String
  price           Float
  createdAt       DateTime   @default(now())
  updatedAt       DateTime   @updatedAt
  orders          Order[]
}
```

Эти модели представляют три таблицы, которые вы ранее спроектировали, включая их связи. Создание моделей в Prisma похоже на описание объектов или типов в TypeScript. Они используют специальный синтаксис Prisma для типов, но он

близок к привычным типам. Ключевой аспект — определение связей между таблицами:

- в `Product` указан `userId`;
- в `User` есть массив `orders`.

Так Prisma описывает связи между таблицами. Для сложных связей рекомендуется изучить документацию.

Для удобства разработки можно использовать плагин Prisma для VS Code (https://oreil.ly/cKsig). Хотя для этого приложения текущих моделей достаточно.

На этом схема данных готова — осталось применить ее к базе через миграцию.

Написание миграций

Миграции — это SQL-запросы, генерируемые ORM на основе определения вашей схемы. При подключении к базе данных для выполнения миграции вы фактически выполняете SQL-инструкции. Именно это делает ORM-инструменты столь полезными. Вместо ручного написания SQL для множества таблиц вы можете описать запрос на TypeScript, а ORM преобразует его в SQL. Разработчики на JavaScript используют эти инструменты, чтобы не изучать все тонкости SQL и особенности работы с базами данных. Запуск миграции также позволяет проверить корректность подключения базы данных к бэкенду. Для запуска миграции в Prisma откройте консоль, перейдите в корень проекта и выполните команду:

```
prisma migrate dev --name initialize_dashboard_db

Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": PostgreSQL database "dashboard", schema "public" at
  "localhost:5432"

Applying migration `20230318132006_initialize_dashboard_db`

The following migration(s) have been created and applied from new schema changes:

migrations/
  └─ 20230318132006_initialize_dashboard_db/
    └─ migration.sql

Your database is now in sync with your schema.
```

После успешной миграции проверьте экземпляр Postgres. В базе данных dashboard должны появиться таблицы с колонками, определенными в ваших моделях. Также вы увидите таблицу `_OrderToProduct`, которая описывает связь между таблицами `Order` и `Product`, заданную в модели. Это самый быстрый способ убедиться в работоспособности подключения. При отсутствии соединения миграция завершилась

бы ошибкой, но при прямом обращении к базе данных можно увидеть, что все в порядке.

При каждом обновлении схемы вам потребуется создавать новую миграцию для обновления базы данных. Обязательно давайте ей описательное имя, чтобы быстро понимать, что произошло в истории изменений БД. Это особенно полезно при возникновении проблем с данными на бэкенде, так как позволяет легко отследить, какие изменения были внесены и когда. В Prisma каждая миграция автоматически генерирует временную метку в начале имени папки. Эта метка важна для определения порядка выполнения миграций при первоначальной настройке базы данных.

Другие инструменты обрабатывают миграции иначе. Например, Knex.js позволяет писать миграции на чистом SQL и генерирует файлы вместо папок. В директории prisma вашего проекта вы увидите папку migrations с подпапкой — здесь хранится сгенерированный SQL для миграций. В этом преимущество ORM: вы пишете, используя знакомый синтаксис, а система генерирует SQL за вас. Изучите файл `migration.sql` в папке миграций:

```
-- CreateTable
CREATE TABLE "User" (
    "id" SERIAL NOT NULL,
    "email" TEXT NOT NULL,
    "name" TEXT NOT NULL,
    "address" TEXT NOT NULL,

    CONSTRAINT "User_pkey" PRIMARY KEY ("id")
);
…
```

Это лишь начальная миграция для настройки БД. По мере роста приложения вы будете добавлять таблицы, колонки и изменять их названия. Если возникнут проблемы с миграцией, ее можно откатить — см. документацию Prisma (https://oreil.ly/PogrK). Сейчас у вас есть все необходимое для добавления начальных данных и продолжения работы над приложением.

Заполнение базы данных начальными данными

Поскольку это ваша локальная база данных, вам понадобятся начальные данные для работы. Вы можете добавить их как часть настройки dev `.env` и использовать при настройке реальной базы данных. Для начала сформируйте файл `seed.ts` в директории prisma. Здесь вы будете использовать Prisma Client для создания записей. Добавьте следующий код в файл `seed.ts`:

```
// seed.ts
const { PrismaClient } = require('@prisma/client');
const db = new PrismaClient();

const main = async () => {
```

```
  const orderData = [...Array(10)].map(() => ({
    id: faker.number.int({ min: 10, max: 170 }),
    total: faker.number.float({ min: 7, max: 15657, precision: 0.01 }),
    createdAt: faker.date.anytime(),
    updatedAt: faker.date.anytime(),
    userId: 5,
  }));

  const productData = [...Array(10)].map(() => ({
    id: faker.lorem.word(),
    name: faker.commerce.productName(),
    price: faker.number.float({ min: 35, max: 1055, precision: 0.01 }),
    createdAt: faker.date.anytime(),
    updatedAt: faker.date.anytime(),
  }));
  …
```

Полный файл `seed.ts` доступен на GitHub (https://oreil.ly/sb7vL).

Этот код позволяет подключиться к базе данных, вставить новые строки в соответствующие таблицы, а затем отключиться. Чтобы запустить это через Prisma, добавьте конфигурацию seed в `package.json`:

```
"collectCoverageFrom": [
      "**/*.(t|j)s"
    ],
    "coverageDirectory": "../coverage",
    "testEnvironment": "node"
  },
  "prisma": {
    "seed": "tsx prisma/seed.ts"
  }
}
```

Это указывает Prisma, где искать файл seed и как его выполнять. После настройки конфигурации и подготовки данных выполните команду для вставки данных в вашу Postgres-базу:

```
npx prisma db seed

Environment variables loaded from .env
Running seed command `ts-node prisma/seed.ts` ..

🌱 The seed command has been executed.
```

Проверив таблицы Postgres (см. рис. 3.6), вы увидите значения из `seed.ts`. На этом работа с данными пока завершена. Для дополнительной проверки используйте этот чек-лист, когда закончите настройку.

- Соответствует ли схема дизайну и функциональности, описанным в документации по поведению (behavioral doc)?
- Проверял ли схему хотя бы один разработчик, кроме вас?

- Проверена ли совместимость схемы со всеми приложениями, использующими эту базу данных?

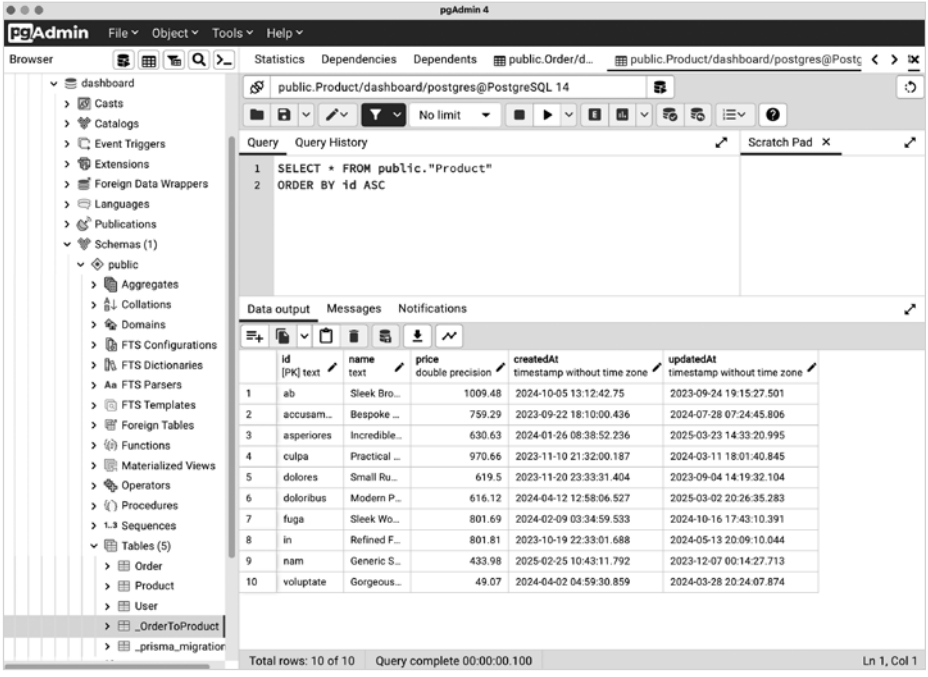

Рис. 3.6. База данных dashboard с начальными данными в локальной Postgres

Существуют и более сложные вопросы, выходящие за рамки этой книги. Вот некоторые из них со ссылками на дополнительные материалы.

- Оставляет ли ваша схема пространство для роста приложения?

  Книга Нила Форда, Марка Ричардса, Прамода Садаладжа, Жамака Дехгани (Neal Ford, Mark Richards, Pramod Sadalage), издательство O'Reilly, *«Software Architecture: The Hard Parts»*[1]

- Существует ли возможность аудита действий и пользователей, их инициировавших?

  Статья *What Is an Audit Trail? Everything You Need to Know («Что такое аудит-трейл? Все, что нужно знать»)* (https://oreil.ly/W0s-P, Auditboard).

- Были ли учтены различные уровни ролей пользователей для доступа к операциям с таблицами?

[1] Форд Нил, Ричардс Марк, Садаладж Прамод, Дехгани Жамак. Современный подход к программной архитектуре: сложные компромиссы. — Питер, 2025.

Статья *Role-Based Access Control (RBAC) («Ролевое управление доступом (RBAC)»)* (https://oreil.ly/XNcZv, Imperva).

Эти сложные темы могли бы стать отдельными главами или даже целыми книгами. Существует множество других вопросов, но на данном этапе не стоит углубляться в детали. Если вы можете ответить на эти вопросы, объяснить решения по схеме другому разработчику и продемонстрировать их команде продукта, этого пока достаточно.

Делайте заметки и создавайте тикеты для любых оптимизаций, которые заметите. Это позволит вам вернуться к деталям позже, поскольку они будут задокументированы для учета при планировании спринта. Убедившись, что схема находится в хорошем состоянии, а тикеты по разным эндпоинтам можно разблокировать, приступайте к разработке API, через которое приложения будут получать данные из созданной базы.

Заключение

В этой главе вы изучили процесс создания диаграмм данных, настройки экземпляра Postgres и первоначальной конфигурации ORM. Теперь вы можете уверенно расширять схему данных, так как уже ответили на множество вопросов о будущем приложения.

ГЛАВА 4

REST API

После определения схемы данных пора подумать о том, как именно их представить. Вы создадите API с несколькими эндпоинтами, выполняющими различные операции, такие как получение всех заказов или отправка новых. Это та часть бэкенда, где мы углубимся в технические детали.

Для создания API, которое смогут поддерживать разные разработчики при смене команды, необходимы стандартные соглашения. Обычно они описываются в документе, определяющем единый подход к реализации кода — от правил именования до кодов ошибок и сообщений в ответах.

Такой набор соглашений упрощает выявление отклонений при проверке пул-реквестов для любых изменений кода. Пул-реквесты — это предлагаемые изменения, которые отправляются на проверку другим разработчикам перед слиянием с основной кодовой базой для развертывания. Поскольку для одной функциональности может быть несколько пул-реквестов, соблюдение этих правил критично для поддержания согласованности кода.

В этой главе рассматривается соблюдение соглашений при разработке API, в том числе следующие аспекты:

- согласование форматов данных с фронтендом и другими сервисами, которые с ними работают;
- написание примера соглашений по коду;
- реализация кода для интерфейса, сервиса и контроллера API;
- отслеживание ошибок через логи и кастомные обработчики ошибок (custom error handlers);
- обеспечение валидации данных.

Эти вопросы возникают при разработке API и его эндпоинтов. Существует множество подходов к проектированию API и архитектурных решений, большинство из которых стоят внимания. Как и всегда, выбор зависит от требований проекта и предпочтений команды.

Для данного API мы будем придерживаться следующих соглашений:

- данные будут передаваться и получаться в формате JSON;
- логика эндпоинтов не должна ссылаться на другие эндпоинты для соблюдения принципа разделения ответственности (separation of concerns);

- именование эндпоинтов должно отражать взаимосвязи данных и функциональности (например, `/orders/{orderId}/products`);
- API должно возвращать стандартные коды ошибок и пользовательские сообщения;
- API должно версионировать эндпоинты для плавного управления устареванием (deprecation);
- API должно обрабатывать вычисления, пагинацию, фильтрацию и сортировку на стороне сервера (backend);
- все эндпоинты, принимающие данные, должны проводить валидацию;
- документация эндпоинтов должна обновляться при любых изменениях.

Обеспечение согласованности фронтенда и бэкенда

Между фронтендом, отображающим данные, и бэкендом, обрабатывающим их, всегда существует взаимодействие. Если фронтенду требуется выполнять многочисленные запросы для получения данных, это может замедлить рендеринг представления для пользователя. В то же время избыточный объем данных в ответах бэкенда приводит к сбору и передаче ненужной информации, увеличивая время отклика. Эта дилемма требует поиска баланса.

Как правило, пагинация, фильтрация, сортировка и вычисления должны выполняться на бэкенде. Это обусловлено большей эффективностью обработки данных на серверной стороне по сравнению с загрузкой всех данных на фронтенд и последующими операциями с ними. Полная загрузка данных приложения на клиентскую сторону требуется крайне редко. Обычно технические специалисты совместно разрабатывают подход к обработке данных, обеспечивающий оптимальный пользовательский опыт (UX). Это может означать внедрение микросервиса в архитектуру для улучшения производительности как фронтенда, так и бэкенда. Если определенный эндпоинт вызывается значительно чаще других, стоит изучить целесообразность его выноса в отдельный микросервис и оценить связанные с этим работы.

Один из спорных моментов — способ отправки данных с эндпоинтов во фронтенд. Для фронтенда может быть предпочтительнее получать несколько данных через один эндпоинт: это улучшает производительность и сокращает количество вызовов. Однако если это нарушает границы данных (data boundaries), необходимо проверить обоснованность их объединения. Границы данных обеспечивают разделение ответственности между функциональностями. На рис. 4.1 показан пример разделения вызовов к эндпоинтам Orders и Products.

Помните: фронтенд может фильтровать данные на уровне представления, но любой пользователь способен проверить ответы сети через инструменты разработчика в браузере. Обеспечение безопасности при работе с данными — всегда ответственность бэкенда. Даже если фронтенд реализует некоторые механизмы защиты, пользователи могут обойти UI и напрямую обратиться к эндпоинтам. Мы

рассмотрим вопросы безопасности в главе 8, но это одна из причин, по которой важно соблюдать границы данных.

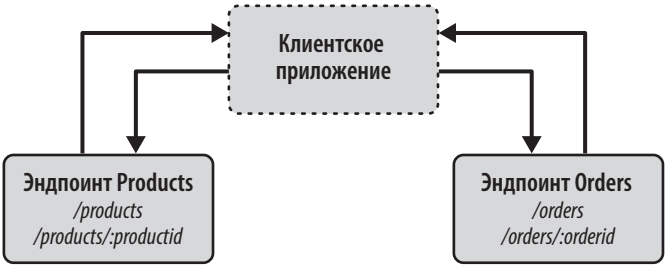

Рис. 4.1. Пример разделения вызовов к эндпоинтам Products и Orders

Теперь, когда требования к фронтенду и бэкенду понятны, перейдем к созданию соглашений для бэкенда.

Создание документа с соглашениями

Такой документ, доступный команде, помогает ужесточить стандарты кода и упрощает адаптацию новых разработчиков к стилю команды. Документ будет дополняться по мере появления новых сценариев. На него команда будет опираться при создании нового эндпоинта или даже целого API — это гарантирует единообразие кода во всех компонентах.

Инструменты вроде Prettier (https://prettier.io), ESLint (https://eslint.org), EditorConfig (https://editorconfig.org) и Husky (https://oreil.ly/qYeav) помогают автоматически соблюдать соглашения в каждом пул-реквесте. Утомительно проверять вручную мелочи вроде пробелов, табуляции, кавычек и тестов. Использование перечисленных инструментов обязывает разработчиков соблюдать соглашения еще до создания коммита. Теперь каждый пул-реквест будет проходить эти проверки, что ускорит ревью и повысит согласованность кода.

Примеры соглашений:

- Microsoft Azure: RESTful Web API Design (https://oreil.ly/uSDNu);
- Google JavaScript Style Guide (https://oreil.ly/CMYVk);
- Airbnb JavaScript Style Guide (https://oreil.ly/M8GTl).

Такие документы могут стать отправной точкой для внутренних соглашений, которые вы адаптируете под предпочтения команды. Например, ваш документ с соглашениями может включать дополнительный раздел, как показано ниже[1]:

[1] В ряде примеров текстовые строки в коде переведены на русский язык для облегчения понимания. На практике в большинстве проектов, особенно в распределенных командах, такие сообщения, как правило, остаются на английском языке. — *Примеч. ред.*

```
Для пагинации ответы должны возвращать такую структуру:
{
 "page": [
    {
      "id": 4,
      "first_name": "Мое имя",
      "last_name": "Моя фамилия",
      "email": "myemail@server.com"
    },
    {
      "id": 5,
      "first_name": "Мое имя",
      "last_name": "Моя фамилия",
      "email": "myemail@server.com"
    },
    {
      "id": 6,
      "first_name": "Мое имя",
      "last_name": "Моя фамилия",
      "email": "myemail@server.com"
    },
],
 "count": 3,
 "limit": 3,
 "offset": 0,
 "total_pages": 4,
 "total_count": 12,
 "previous_page": 1,
 "current_page": 2,
 "next_page": 3
}

Для обработки ошибок ответы должны возвращать такую структуру:
{
  "errors": [
    { "statusCode": "111", "message": "тип значения возраста должен быть int" },
    { "statusCode": "112", "message": "email – это обязательное поле" }
  ]
}
```

Это еще один способ получить обратную связь от коллег и сделать документ с соглашениями максимально полезным для всех.

Некоторые установленные соглашения могут временами раздражать, особенно в условиях жестких сроков. Поэтому важно сначала реализовать несколько функций и проверить их на практике, а уже потом строго внедрять соглашения через линтеры и другие скрипты. Но после того как соглашения будут хорошо проработаны, не отклоняйтесь от них, если только не произойдет существенных изменений в проекте. Именно согласованность во всех кодовых базах упростит их долгосрочную поддержку.

Создание API и первого эндпоинта

Как обсуждалось в главе 2, NestJS — это фреймворк, который мы будем использовать. Не разбирая подробно, как писать весь код, сосредоточимся на фундаментальных причинах выбора конкретных способов реализации функциональности. Эта логика применима к любому фреймворку. Для понимания синтаксиса кода самостоятельно изучите документацию NestJS.

При разработке бэкенда необходимо учитывать множество аспектов. Секрет в том, чтобы сначала выбрать одно направление и сфокусироваться на нем. Позже вы вернетесь к вопросам безопасности, производительности и тестирования. Сейчас же важно обеспечить работоспособность эндпоинтов, чтобы фронтенд мог начать интеграцию UI с API. Начните с реализации базовых CRUD-операций (Create, Read, Update, Delete), которые потребуются приложению. В этом проекте понадобятся эндпоинты для:

- управления товарами (Products);
- управления заказами (Orders);
- административных функций.

Первые две группы эндпоинтов основаны на уже известных требованиях приложения. Третья группа появится после обсуждений с продуктовой командой. Некоторые функции потребуются только службе поддержки и должны быть недоступны обычным пользователям.

Работа с разными уровнями прав доступа и система контроля доступа рассмотрены в главе 8. Помните: эндпоинты, вероятно, будут изменяться. Но нам нужно с чего-то начинать.

Вы можете удалить некоторые шаблонные файлы, а именно `app.controller.spec.ts`, `app.controller.ts` и `app.service.ts`. Также обновите `app.module.ts`, удалив ссылки на эти файлы. Это поможет поддерживать чистоту кода по мере внесения изменений и добавления нового функционала. Эти файлы служат лишь примерами реализации функциональности эндпоинтов. Удаление части примеров при использовании инструментов каркаса (scaffolding tools) — стандартная практика, позволяющая оставить только необходимое. С опытом вы научитесь определять, какие файлы можно удалить, а какие являются полезной отправной точкой.

Создание эндпоинтов для заказов

Начните с реализации функциональности для заказов. В директории `src` создайте подпапку `orders` и добавьте файлы для обработки типов данных, тестов, сервиса и контроллера этого эндпоинта.

Файл `orders.controller.ts` содержит определения всех эндпоинтов для этой функциональности. Подробнее о работе контроллеров можно узнать в документации

NestJS (https://oreil.ly/X_2ET). Если будет нужно получить или изменить заказы, фронтенд обратится к этим эндпоинтам. Это оптимальное место для первичной валидации данных из запросов. Пример эндпоинта:

```
// orders.controller.ts
…
@Get()
  public async orders(): Promise<Array<Order>> {
    try {
      const orders = await this.ordersService.orders({});
      return orders;
    } catch (err) {
      if (err) {
        throw new HttpException('Not found', HttpStatus.NOT_FOUND);
      }
      throw new HttpException('Generic', HttpStatus.BAD_GATEWAY);
    }
  }
```

В примере показана кастомная обработка ошибок с отправкой сообщения и статус-кода, соответствующих соглашениям. Контроллер не должен содержать бизнес-логику — она обрабатывается в сервисе. Контроллеры отвечают только за обработку запросов и ответов, что улучшает тестируемость кода и обеспечивает четкое разделение обязанностей.

Контроллеры также выполняют часть валидации, о которой упоминалось выше. Рассмотрим эндпоинт для обновления заказов:

```
// orders.controller.ts
…
@Patch(':id')
public async update(
  @Param('id', ParseIntPipe) id: number,
  @Body() order: UpdateOrderDto,
): Promise<Order> {
  try {
    return await this.ordersService.updateOrder({
      where: { id },
      data: order,
    });
  } catch (err) {
    if (err) {
      throw new HttpException('Not found', HttpStatus.NOT_FOUND);
    }
    throw new HttpException('Generic', HttpStatus.BAD_GATEWAY);
  }
}
```

Обратите внимание на использование `try-catch` для обоих эндпоинтов. Это хороший способ гарантировать обработку ошибок:

- код в блоке `try` выполняется первым;

- при ошибках срабатывает блок `catch`;
- далее можно определить логику обработки ошибки.

Такой подход часто упускают в спешке, поэтому его стоит включить в соглашения по коду. Валидация осуществляется через `UpdateOrderDto`. Пример из `orders.interface.ts`:

```
// orders.interface.ts
…
export class UpdateOrderDto {
  @IsNumber()
  total: number;

  @IsNotEmpty()
  products: Product[];

  @IsNotEmpty()
  userId: number;
}
```

DTO (*Data Transfer Object*) означает объект передачи данных. DTO используются для инкапсуляции данных, которые часто применяются сервисным слоем бэкенда, чтобы уменьшить объем передаваемых данных между бэкендом и фронтендом. В MVC-фреймворках, таких как NestJS, они также служат моделями приложения. В основном они определяют параметры, передаваемые в методы бэкенда.

NestJS выполняет валидацию «под капотом» с помощью пакета (https://oreil.ly/A2p_f) `class-validator` (https://oreil.ly/bbyFa), поэтому, если `userId` пуст или `total` не является числом, на фронтенд будет отправлена ошибка о неверном формате данных. Корректные сообщения валидации помогают разработчикам понять, что делать, и предоставляют пользователям полезную информацию.

Разработка сервиса заказов

Последний файл — `orders.service.ts`. В нем реализована бизнес-логика для функциональности заказов. Подробнее о сервисных файлах см. в документации NestJS (https://oreil.ly/s2jZ6). Все вычисления, сортировка, фильтрация или другие манипуляции с данными происходят в этом файле. Пример метода для обновления заказа:

```
// orders.service.ts
…
public async updateOrder(params: {
  where: Prisma.OrderWhereUniqueInput;
  data: Prisma.OrderUpdateInput;
}): Promise<Order> {
  const { data, where } = params;
  this.logger.log(`Обновление существующего заказа ${where.id}`);

  try {
```

```
    const updatedOrder = await this.prisma.order.update({
      data: {
        ...data,
        updatedAt: new Date(),
      },
      where,
    });

    this.logger.log(`Успешное обновление заказа ${updatedOrder.id}`);

    return updatedOrder;
  } catch (err) {
    this.logger.log(`Ошибка обновления заказа ${where.id}`);

    throw new HttpException(err.message, HttpStatus.CONFLICT);
  }
}
```

Итак, мы применили отличный метод для бэкенда и смогли соблюсти соглашения, поскольку здесь реализованы обработка ошибок и логирование (https://oreil.ly/k3F8n). Ошибки также могут возникать на уровне сервиса, поэтому используется конструкция `try-catch` для передачи этих ошибок обратно в контроллер.

Логирование следует добавить и в контроллеры. Вы убедитесь, что логи незаменимы при отладке бэкенда. Делайте записи в логах максимально описательными, чтобы отслеживать значения в базе данных, других эндпоинтах или сторонних сервисах.

Независимо от выбранного фреймворка для бэкенда, ключевые элементы остаются неизменными: это валидация входных данных, логирование критических участков потока и обработка потенциальных ошибок. Соблюдение этих принципов и соглашений по коду приведет команду к созданию надежной кодовой базы.

Теперь стоит рассмотреть другие аспекты бэкенда, которые гарантируют корректную работу эндпоинтов и сервисов.

Проверка подключения к базе данных

Во многих проектах есть папка с названием utils, helpers или подобным. Вам потребуется создать такую папку для размещения сервиса, который будет использоваться для запуска инстанса Prisma Client и подключения к базе данных. Мы обсуждали это в главе 3, и теперь будем расширять уже имеющуюся основу. В папке src создайте новую папку utils. В этой папке создайте файл `prisma.service.ts` и поместите в него код из документации NestJS (https://oreil.ly/CzD1B).

В большинстве случаев не нужно писать все с нуля, если потратить несколько минут на изучение документации. Это то, что senior-разработчики делают постоянно. Также не бойтесь добавлять другие элементы в папку utils! Если вы видите небольшие функции, которые повторяются во многих частях приложения (например,

форматтеры данных), переносите их сюда, чтобы другим разработчикам было проще их найти и использовать.

Если вы еще не сделали коммит в Git, сейчас подходящий момент. Теперь бэкенд в том состоянии, когда другие разработчики могут подключиться и добавить новую функциональность или конфигурации. Одна из самых сложных задач — подготовить основу для улучшений другими. Именно этим вы сейчас и занимаетесь.

По мере развития приложения в рамках книги вы начнете добавлять вызовы сторонних сервисов, обрабатывать данные из различных источников и работать с вопросами безопасности. Все это будет включать эндпоинты и другие методы сервисов, которые вы внесете по мере выполнения задач в спринте.

Важно получить практический опыт, поэтому попробуйте добавить обработку ошибок, логирование и валидацию к оставшимся эндпоинтам в контроллере заказов. Разумеется, вы также можете обратиться к GitHub-репозиторию (https://oreil.ly/tbcvR).

Заключение

В этой главе мы рассмотрели множество аспектов, в которые погрузимся еще глубже в 5–10-й главах! Ключевые выводы о том, что вам нужно делать:

- достигать договоренностей с разработчиками фронтенда;
- устанавливать строгие соглашения по коду для API как можно раньше;
- создавать начальные эндпоинты для старта работы фронтенд-разработчиков;
- обрабатывать, валидировать и логировать ошибки.

Теперь, когда у нас есть несколько рабочих эндпоинтов, пора расширить их функциональность.

ГЛАВА 5

Сторонние сервисы

В процессе разработки продукта наступит момент, когда вам потребуется использовать сторонний сервис. Сторонние сервисы предоставляют сложную или проприетарную функциональность, которая существует вне вашей кодовой базы и компании. Сторонний сервис обычно управляется другой организацией; вы оплачиваете его использование и получаете доступ к поддержке при возникновении проблем.

Сторонние сервисы актуальны, когда вам требуется функциональность, реализация которой заняла бы значительное время, была бы сложна в поддержке и, вероятно, потребовала бы отдельной команды, например:

- платежные системы;
- аутентификация/авторизация;
- мониторинг и логирование для вашего приложения.

В этой главе мы рассмотрим:

- выбор сторонних сервисов;
- интеграцию стороннего сервиса с нашей системой;
- управление ошибками, простоями и обновлениями.

Работа со сторонними сервисами неизбежна, поэтому важно заранее понимать, с чем вы столкнетесь. Обычно такие сервисы предоставляют API или SDK в виде npm-пакета для установки в приложение, но для доступа к полному функционалу потребуются учетные данные. Некоторые SDK существуют автономно (без привязки к сервису) и предлагают узкоспециализированные возможности.

Помните: изменения на стороне сервиса могут привести к неожиданным сбоям без предупреждения. Поскольку вы работаете только с интерфейсным кодом, это стандартный риск для подобных интеграций.

Мы добавим Stripe в качестве стороннего сервиса для обработки платежей в приложении. Сначала рассмотрим ключевые аспекты выбора сторонних сервисов. Затем вы реализуете контроллер, сервисный слой и остальной код для интеграции Stripe в демопроект.

Выбор стороннего сервиса

Существует множество сервисов для обработки платежей, аутентификации, работы с данными, налоговых расчетов, логирования, интеграции со сторонними системами и других функций. Это может сделать выбор сложной задачей, поскольку требуется решить, где затраты выше: при собственной разработке или при приобретении чужого сервиса. По сути, процесс аналогичен выбору пакета для вашего кода.

Ключевое отличие выбора стороннего сервиса от пакета — платность сервисов. Нет ничего плохого в выборе платного сервиса вместо бесплатного пакета, если это сэкономит время разработки, ускорит выход функций и повысит уверенность в работоспособности приложения. Попасть в экосистему качественного продукта с прозрачным ценообразованием гораздо лучше, чем не попасть.

Вот на что стоит обратить внимание при анализе сервисов.

- Есть ли хорошо проработанная документация, которая регулярно обновляется?
- Как осуществляется выпуск новых версий?
- Доступна ли оперативная поддержка?
- Четко ли определена структура ценообразования, даже если потребуется созваниваться с торговым представителем?
- Есть ли альтернативные варианты для нужной вам функциональности?
- Насколько компания зрелая и стабильная?
- Легко ли интегрировать решение в существующую архитектуру?
- Как это решение выглядит в сравнении с популярными аналогами?
- Есть ли хорошая песочница (sandbox) для тестирования?
- Какие усилия потребуются для перехода на другой инструмент?

Это все важные аспекты для рассмотрения, и здесь будут интересные компромиссы. Возможно, вы даже углубитесь в отраслевые вопросы соответствия законодательству и нормативным требованиям. После тщательного бизнес-анализа сервиса протестируйте его в своей кодовой базе. Засеките, сколько времени займет добавление минимально рабочей интеграции, — это будет хорошей предварительной проверкой удобства работы с сервисом.

При проверке не забывайте делиться результатами с командой и руководством. Проводите краткие демонстрации, чтобы показать прогресс в коде. Это поможет коллегам повысить квалификацию, а вам — глубже разобраться в сервисе, отвечая на их вопросы.

Параллельно с интеграцией изучите репутацию продукта и компании. Даже если условия кажутся вам идеальными, поинтересуйтесь опытом работы других разработчиков с сервисом, особенно после того, как они подписали контракт. Это

может выглядеть несколько избыточным, но критически важно проверить все до глубокой интеграции.

Учитывайте страну происхождения сервиса. Для государственных проектов некоторые страны могут быть под запретом. Геополитика влияет на инструменты разработки, даже если это неочевидно в вашей отрасли. И наоборот: если компания хочет поддержать конкретную страну, это может стать аргументом при выборе сервиса.

Продолжая исследовать сторонние сервисы, изучите их репозитории на GitHub. Это позволит увидеть опыт других разработчиков, а также то, как компания — провайдер сервиса реагирует на баги и вопросы. Так вы заранее обнаружите типичные проблемы, с которыми сталкиваются пользователи, прежде чем углубитесь в интеграцию.

Также проверьте, как сторонние сервисы взаимодействуют между собой. Возможно, вам потребуется использовать несколько сервисов через промежуточные решения — например, Zapier (https://zapier.com) или сервисы облачного провайдера, — что усложнит архитектуру. Оцените, насколько они дополняют уже используемые вами инструменты.

Список потенциальных сервисов

Выбор стороннего сервиса может быть сложным. Вот отправные точки для поиска.

Обработка платежей:

- Stripe (https://stripe.com/docs);
- Square (https://developer.squareup.com/docs);
- Clover (https://docs.clover.com/docs);
- PayPal (https://developer.paypal.com/home);
- Paddle (https://www.paddle.com).

Логирование и мониторинг:

- DataDog (https://docs.datadoghq.com);
- New Relic (https://docs.newrelic.com);
- Splunk (https://docs.splunk.com/Documentation);
- Sentry (https://docs.sentry.io).

Сторонние приложения:

- Meta (https://developers.facebook.com/docs);
- Instagram (https://developers.facebook.com/docs/instagram);
- YouTube (https://developers.google.com/youtube/v3);
- Google Workspace (https://developers.google.com/workspace).

Электронная коммерция:

- Shopify (https://shopify.dev/docs/api);
- Amazon (https://developer-docs.amazon.com/sp-api);
- Etsy (https://developers.etsy.com/documentation);
- BigCommerce (https://developer.bigcommerce.com/api-docs/overview).

Аутентификация

- Auth0 (https://auth0.com/docs);
- FusionAuth (https://fusionauth.io/docs);
- Amazon Cognito (https://docs.aws.amazon.com/cognito);
- SuperTokens (https://supertokens.com/docs/guides);
- Clerk (https://clerk.com).

Электронная почта:

- SendGrid (https://docs.sendgrid.com/for-developers);
- Amazon Simple Email Service (SES) (https://docs.aws.amazon.com/ses/latest/dg/send-email-api.html);
- Mailgun (https://documentation.mailgun.com/en/latest/quickstart.html);
- Postmark (https://postmarkapp.com/email-api);
- Brevo (https://www.brevo.com).

Геолокация:

- Google Maps (https://developers.google.com/maps/documentation/javascript);
- Mapbox (https://www.mapbox.com);
- Esri ArcGIS (https://developers.arcgis.com/javascript/latest);
- Radar (https://radar.com/documentation).

В зависимости от потребностей вашего продукта вы можете столкнуться с очень узкоспециализированными сервисами. Их интеграция может быть сложной, поэтому важно четко документировать находки и вовлекать коллег в поиск.

Всегда проверяйте структуру данных в ответах API — некоторые из них плохо документированы. Проведите предварительное тестирование сервисов, чтобы понять, какие данные они возвращают. Этот подход также помогает оценить совместимость сервиса с вашей инфраструктурой. Структуры ответов некоторых сервисов могут вас удивить.

Так как ваше приложение требует обработки платежей, вам понадобится сторонний сервис для этой функции. Такой сервис должен соответствовать стандарту

PCI DSS (Payment Card Industry Data Security Standard) и обеспечивать защиту финансовых и персональных данных пользователей. В данном проекте вы будете работать со Stripe.

Интеграция Stripe

Stripe для этого проекта выбран потому, что у него отличная документация (к которой вы будете часто обращаться), его использует большое сообщество разработчиков и там есть довольно хорошая тестовая среда. Тем не менее я рекомендую вам изучить и другие варианты и оценить плюсы и минусы.

Я не буду подробно описывать настройку аккаунта Stripe, так как это довольно глубокая тема и вам потребуется ввести личные данные. Если вам это неудобно — ничего страшного. В любом случае, эту информацию предоставит ваша организация. Я сосредоточусь на программной реализации, чтобы вы могли следить за кодом и понимать, что делать после настройки аккаунта.

Первое, что нужно сделать, — решить, как вы будете работать со сторонними сервисами в вашем приложении. Учитывайте, что по мере роста продукта и добавления новых функций интеграций станет больше. В этом проекте мы создадим папку integrations внутри src, а в ней — подпапку stripe с несколькими файлами.

```
|__ prisma
|__ src
|____ integrations
|______ stripe
|________ stripe.controller.spec.ts
|________ stripe.controller.ts
|________ stripe.interface.ts
|________ stripe.module.ts
|________ stripe.service.spec.ts
|________ stripe.service.ts
|____ orders
|____ app.module.ts
|____ main.ts
|__ test
```

Это также отличный момент для обновления архитектурной диаграммы, чтобы показать, как новый сервис интегрируется в ваше приложение (рис. 5.1). Документация никогда не бывает по-настоящему завершенной, поэтому не забывайте обновлять соответствующие документы при добавлении нового функционала. Взаимодействие сторонних сервисов с приложением включает обработку событий через облачный сервис с использованием инструментов вроде Amazon CloudWatch (https://oreil.ly/16lJQ) и EventBridge (https://oreil.ly/2rJzv), а также прямое взаимодействие с вашими API.

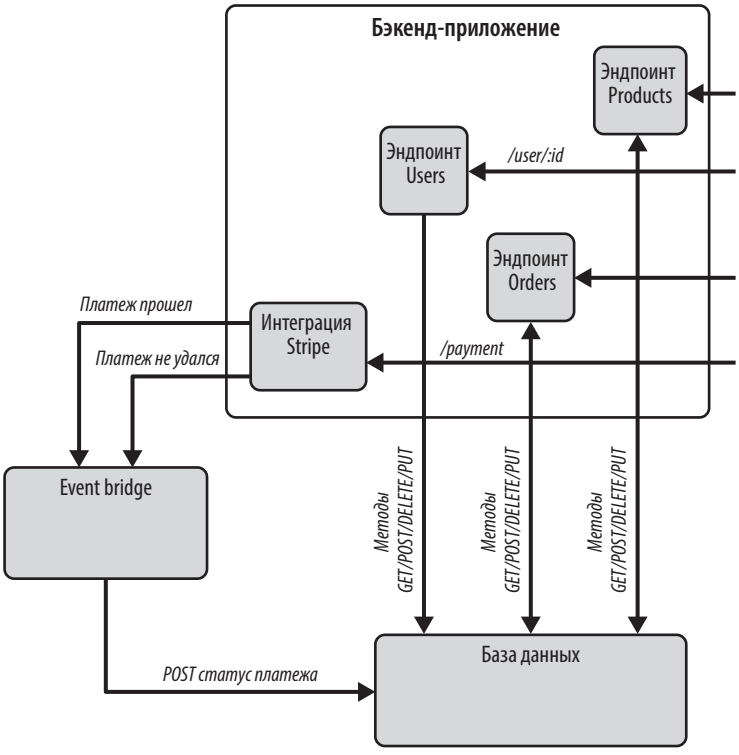

Рис. 5.1. Обновленная архитектурная диаграмма

Здесь вы будете использовать тот же шаблон программирования, что и для функционала заказов. Эндпоинты, вызываемые фронтендом, определяются в контроллере. Взаимодействие с API Stripe будет находиться в сервисе. Такое разделение ответственности делает код модульным и упрощает тестирование. Этот подход следует сохранять даже при добавлении другого стороннего кода.

Можно начать с обновления файла `stripe.module.ts`, добавив импорт и использование нового сервиса и контроллера, — это даст быстрый результат:

```
// stripe.module.ts
import { Module } from '@nestjs/common';
import { StripeService } from './stripe.service';
import { StripeController } from './stripe.controller';

@Module({
  exports: [StripeService],
  providers: [StripeService],
  controllers: [StripeController],
})
export class StripeModule {}
```

Начать с «быстрой победы» — хороший способ создать импульс при работе над крупной задачей. Это делает проблему менее пугающей. Реализуя функциональность, обратите внимание, предпочитаете ли вы писать код в определенном порядке. Это может ускорить переход к сложным деталям. В этой книге я обычно начинаю с импортов модулей, перехожу к контроллеру и завершаю сервисом. Интерфейсы обновляются по ходу работы, а тесты оставляю на потом. Однако в реальной разработке я *часто* переключаюсь между файлами.

Для работы с Stripe API необходимо добавить новую переменную окружения в файл `.env`. Это значение можно получить из вашего аккаунта, но также доступен тестовый ключ, указанный в документации. Добавьте в `.env` следующую строку:

```
STRIPE_SECRET_KEY="sk_your_stripe_account_secret_key"
```

Разработка контроллера

Теперь можно перейти к Stripe-контроллеру и начать создавать эндпоинты. Потребуются:

- эндпоинт для приема платежей от пользователей;
- эндпоинт для обновления товаров в системе Stripe.

Откройте `stripe.controller.ts` и добавьте следующий код для создания эндпоинтов:

```
// stripe.controller.ts
import {
Body,
Controller,
Headers,
HttpException,
HttpStatus,
Post,
Res
} from '@nestjs/common';
import { StripeService } from './stripe.service';
import { CreateStripePaymentDto } from './stripe.interface';

@Controller('/v1/stripe')
export class StripeController {
  constructor(private readonly stripeService: StripeService) {}

  @Post('/payments')
  public async createPayment(
    @Body() payment: CreateStripePaymentDto,
    @Headers() headers,
    @Res() res,
  ) {
    try {
      const paymentInfo = await this.stripeService.createPayment({
```

```
        payment,
        res,
        origin: headers.origin,
      });

      return paymentInfo;
    } catch (err) {
      throw new HttpException('Произошла ошибка', HttpStatus.NOT_FOUND);
    }
  }
}
```

В данном контроллере определен один эндпоинт. Он обрабатывает платежи через Stripe и имеет особенность: в методе сервиса требуется выполнить редирект, так как пользователь будет перенаправлен на страницу оплаты Stripe через платежный запрос, а затем необходимо вернуть его обратно в приложение после завершения платежа. Принцип работы редиректа можно изучить на схеме процесса оплаты Stripe (https://oreil.ly/WuKg-).

Учтите, что в контроллере используется `CreateStripePaymentDto`, поэтому необходимо убедиться, что интерфейс соответствует точным требованиям к данным. По мере роста приложения понадобится вложенная валидация, чтобы гарантировать корректность передаваемых в API Stripe значений:

```
// stripe.interface.ts
import { IsNotEmpty, MinLength } from 'class-validator';

export class CreateStripePaymentDto {
  @IsNotEmpty()
  priceId: string;

  @MinLength(1, {
    message: 'Количество должно быть не менее 1',
  })
  quantity: number;
}
```

Если вы не используете NestJS, то функцию валидации можно реализовать в вашем промежуточном ПО (middleware) с помощью других пакетов: например, Joi (https://joi.dev) или Zod (https://oreil.ly/tXs42). `Stripe PriceData` — это то, что вы, вероятно, будете применять многократно по мере развития вашей сервисной реализации. Хорошей практикой является копирование типов и требований из документации сервиса, чтобы гарантировать отправку именно тех данных, которые ожидаются. Иногда можно просто использовать типы, предоставляемые непосредственно SDK.

Бывают случаи, когда вы знаете, что будете использовать только определенные поля из типов SDK и добавлять другие поля, которых нет в SDK. Вам нужно решить, стоит ли тратить время на адаптацию типов SDK под свои нужды или лучше создать собственные типы. Если вы уверены, что будете применять ответ в том виде, в каком он приходит, смело используйте типы SDK; если же вам нужна новая

комбинация полей или другой формат типов, рассмотрите возможность создания собственного интерфейса путем копирования типов.

Не все сервисы столь же дружелюбны к разработчикам, как Stripe. Если вы не уверены, какие типы ожидаются в запросе или что вернется в ответе, просто протестируйте сервис и посмотрите результат. Иногда без этапа изучения не обойтись, чтобы понять, с чем придется работать и как добавить корректные типы в приложение. Для этого можно использовать инструменты вроде mitmproxy (https://mitmproxy.org) или даже Postman.

Разработка сервиса

После реализации основной части кода контроллера переключайтесь на сервис для прямой работы со Stripe. Stripe предоставляет удобный npm-пакет для взаимодействия с API.

Давайте реализуем функционал платежей в Stripe. Во время написания кода можно обращаться к документации Checkout API (https://oreil.ly/M-6g5). В файле `stripe.service.ts` выполним базовую настройку: импорт пакетов, добавление конструктора и логгера. Затем добавим метод для обработки запросов к Checkout API Stripe:

```
// stripe.service.ts
…

@Injectable()
export class StripeService {
  private readonly logger = new Logger(StripeService.name);
  // Версия строки оставлена такой, потому что этого требует Stripe
  private stripe = new Stripe(process.env.STRIPE_SECRET_KEY, {
    apiVersion: '2023-08-16'
  });

  public async createPayment({
    payment, origin, res
  }: {
    payment: CreateStripePaymentDto;
    origin: string;
    res: any;
  }) {
    this.logger.log('Начало платежа в Stripe');

    try {
      // Создаем сессию Checkout с параметрами из тела запроса
      const session = await this.stripe.checkout.sessions.create({
        line_items: [{
          // Указываем Price ID (например, price_H5ggYwtDq4fbrJ) товара
          price: payment.priceId,
          quantity: payment.quantity,
        }],
        mode: 'payment',
```

```
        success_url: `${origin}/?success=true`,
        cancel_url: `${origin}/?canceled=true`,
      });

      res.status(303).redirect(session.url);
    } catch (err) {
      throw Error('Произошла ошибка в Stripe');
    }
  }
}
```

Одним из ключевых моментов здесь является логгер (logger), инициализированный в сервисе. Записываемые логи будут бесценны при отладке, так как позволяют фиксировать каждое действие. При интеграции со сторонними сервисами, такими как Datadog, эти данные помогут точно диагностировать проблемы.

Самое главное при интеграции сторонних сервисов — это логирование и обработка ошибок: ваш код должен корректно обрабатывать любые сбои в стороннем коде. Важно учитывать, что такой код может нарушать ваши внутренние соглашения, но это неизбежный компромисс при использовании внешних сервисов.

Также стоит заранее продумать возможные форс-мажорные случаи. Например, как поведет себя приложение при недоступности сервиса? Эти вопросы необходимо обсуждать с продуктовой командой по мере выявления особенностей сервиса в ходе тестирования — подробнее об этом мы поговорим в главе 7.

Вы увидите, что в базу данных необходимо добавить новое поле: `stripeProductId`. Вам нужно отслеживать некоторые сторонние данные в своей системе, чтобы выполнять действия через API и проверять изменения. Это означает, что вам потребуется обновить `schema.prisma` и выполнить миграцию. Я оставлю эту задачу вам, но вы можете сверить свой файл `schema.prisma` с версией в GitHub-репозитории (https://oreil.ly/4mlvc).

Заключение

Эта глава была посвящена тому, как анализировать и использовать сторонние сервисы в своем коде. Самое важное при работе с такими сервисами — помнить, что иногда они ведут себя не так, как вы ожидаете. Может возникать непредсказуемый порядок выполнения (race conditions), данные могут возвращаться в неожиданных форматах, а параметры запросов — меняться. Здесь вашими лучшими друзьями станут обработка ошибок и логирование. Время от времени будут возникать проблемы со сторонними сервисами, и это нормально. С ними можно справиться благодаря коммуникации между командой разработки и продуктовой командой, а также готовности пробовать разные стратегии кода.

ГЛАВА 6

Фоновые задачи

После интеграции сторонних сервисов часто возникает необходимость в автоматизированных способах синхронизации данных компании с данными стороннего сервиса, а также в запуске других событий, таких как отправка электронных писем. Именно здесь задействуются фоновые и cron-задачи.

Фоновая задача (*background job*) выполняет код или действия, которые должны систематически выполняться вне потока API. Сюда входят такие задачи, как отправка электронных писем или выполнение сложных вычислений после обновления данных пользователем. Фоновые задачи работают параллельно с кодом приложения на сервере и обычно запускаются напрямую из вызова эндпоинта. Эти задачи чаще всего определяются на уровне сервиса, поскольку они, как правило, содержат бизнес-логику.

Cron-задача (*cron job*) — это операция, выполняемая по расписанию. Она запускается с заданным интервалом времени в соответствии с бизнес-требованиями. Синхронизация данных со сторонним сервисом или получение данных от него по расписанию — типичный пример использования cron-задачи. Хотя такие задачи могут выполняться в вашем приложении, более распространенной и правильной практикой является их вынесение на отдельные ресурсы, например на экземпляр Amazon Elastic Compute Cloud (EC2) или другой сервер. Такой подход позволяет не расходовать ресурсы, используемые приложением.

Основная сложность работы с фоновыми и cron-задачами заключается в том, что из-за автономного выполнения по расписанию они могут непредсказуемо влиять на состояние всей системы. Именно здесь проявляются навыки senior-разработчика.

В этой главе мы рассмотрим:

- интеграцию фоновых задач и cron-задач в архитектуру вашего приложения;
- настройку оповещений, мониторинга и логирования для фоновых и cron-задач;
- обработку проблем синхронизации данных со сторонними сервисами через cron-задачи;
- решение проблем выполнения задач в фоновых процессах;
- обеспечение долгосрочной поддержки фоновых и cron-задач.

Прежде всего стоит отметить, что задачи могут быть написаны на любом языке программирования. Таким образом, вы не делаете ничего принципиально нового — попрежнему будете использовать TypeScript. Это просто код, который выполняется отдельно от эндпоинтов или бизнес-логики сервиса. Именно поэтому понимание общей архитектуры приложения критически важно: задачи могут теряться со временем, поскольку разработчики приходят и уходят, а продукт эволюционирует.

От вас ожидается умение вносить изменения в архитектуру и документировать их, а также объяснять их влияние на распределение ресурсов. Возможно, вам также придется проводить первоначальное исследование инструментов, которые лучше всего подходят для вашей инфраструктуры. В этом случае крайне полезно обсудить вопросы с командой DevOps или теми, кто управляет инфраструктурой.

Узнайте, какие облачные провайдеры используются для работы со всеми приложениями компании. Это поможет вам понять, какие инструменты доступны и как подходить к написанию кода для ваших задач. Выбранный инструмент напрямую повлияет на реализацию, так как вам придется использовать его SDK для взаимодействия между его ресурсами и вашим кодом. Подробнее об этом мы поговорим далее в этой главе, но прежде чем углубляться в детали, начнем с архитектуры.

Обновление архитектуры бэкенда

В этом проекте вам предстоит реализовать фоновую задачу для отправки электронных писем и cron-задачу для синхронизации данных из Stripe с вашей базой данных. Чтобы вы и вся команда четко понимали, как эти компоненты вписываются в бэкенд, необходимо обновить архитектурную диаграмму. Когда возникают вопросы о взаимодействии компонентов или обосновании использования ресурсов, архитектурная диаграмма должна служить единственным источником истины (source of truth). Крайне важно поддерживать этот документ в актуальном состоянии. Кроме того, диаграмма помогает выявить потенциальные проблемы на раннем этапе интеграции новых фич в бэкенд. На рис. 6.1 показан обновленный вариант диаграммы.

Обновление документации — это прекрасная возможность проверить глубину своих знаний. Одним из ключевых навыков senior-разработчика является наставничество для junior-разработчиков, и его важность сложно переоценить. Используйте обновления архитектуры как способ познакомить команду с новыми концепциями и освежить в памяти принципы работы всего продукта, включая те аспекты, в которых вам самим может потребоваться повторение.

Теперь у вас есть визуальное представление потока выполнения задач. Фоновое задание для отправки электронных писем будет запускаться только при возникновении событий в интеграции со Stripe. Cron-задача для синхронизации данных Stripe (например, товаров) с вашей базой данных будет выполняться согласно заданному графику.

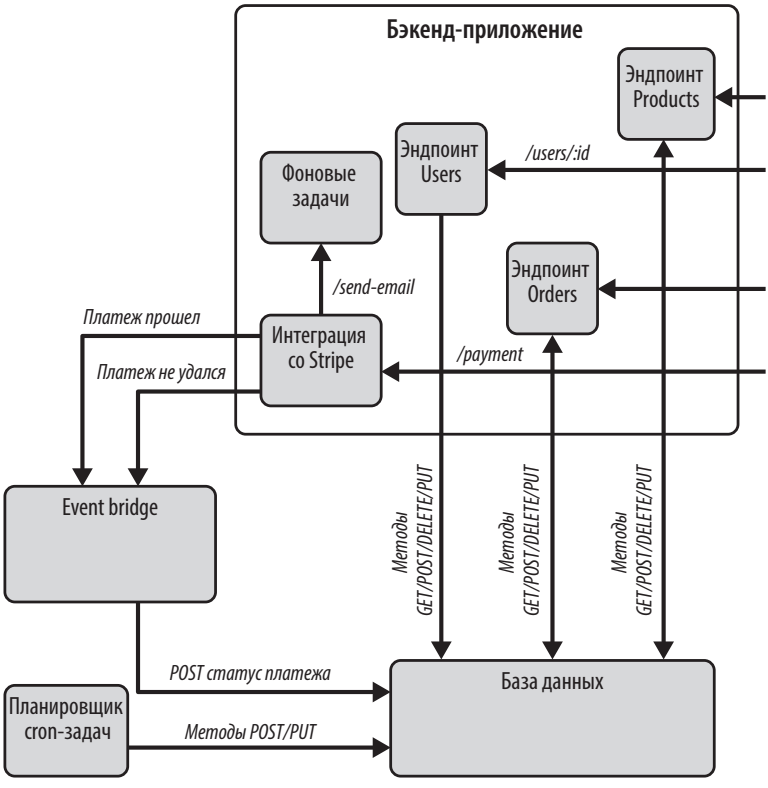

Рис. 6.1. Обновленная архитектурная диаграмма с фоновыми и cron-задачами

Альтернативный взгляд на структурирование cron- и фоновых задач в коде приложения

Итан Браун дает ценные рекомендации по этому вопросу. Вот его мнение.

> В последнее время я отдаю предпочтение монорепозиториям (monorepos) и в большинстве своих проектов размещаю код cron- и фоновых задач вместе с основным кодом приложения. Разумеется, он организован в соответствующие папки, но не вынесен в отдельный репозиторий или проект — и уж точно не является микросервисом. Я рассматриваю это как единый код приложения.
>
> Часть этого кода выполняется как прямой ответ на запросы пользователей; другая часть запускается по расписанию (cron-задачи); третья — непосредственно или опосредованно в результате действий пользователя, но с достаточно длительным временем выполнения, чтобы реализовать меха-

низм оповещения заинтересованных сторон о завершении задачи (фоновые задачи).

В целом этот подход предлагает мощную парадигму, которая помогает прояснить суть cron-задач и фоновых заданий. Стоит рассматривать их как часть предметной области (то есть кода приложения), которая в основном управляется событиями: большинство событий могут быть прямыми запросами пользователей (обычно HTTP, XHR или WebSockets), но одни события запускаются по расписанию, а другие — очередями (в моем случае это SQS (Simple Queue Service) или SNS (Simple Notification Service) от Amazon).

Для фоновых заданий, связанных только с чтением данных (например, аналитикой или отчетностью), я бы сделал исключение из этой парадигмы. Если такое задание не изменяет состояние системы, а лишь отслеживает его или выполняет бизнес-аналитику, его можно вынести в микросервис или отдельное приложение.

Но если задание изменяет состояние системы, это код приложения, независимо от того, как оно было вызвано.

Я применяю этот подход и к системным диаграммам: они должны отображать не только эндпоинты приложения, но и типы событий, которые эти задания запускают.

Теперь, когда у вас есть представление о том, как эти задачи будут работать в существующей архитектуре, самое время взглянуть на инфраструктуру.

Использование cron-задач

Здесь мы погрузимся в довольно специфичные вещи, поэтому убедитесь, что у вас есть обратная связь от команды DevOps. В этом проекте мы предполагаем, что ваша инфраструктура построена на Amazon Web Services (AWS). В книге не будет подробного разбора сервисов, только краткие примеры реализации. Если вы хотите узнать больше о конкретном облачном провайдере, рекомендуется начать с его документации, блогов и бесплатных обучающих материалов.

Планирование cron-задач обычно настраивается через облачные конфигурационные файлы, которые вы, как правило, не будете изменять. Однако вы будете отвечать за код, который выполняет задачу, и должны знать ее расписание. Пример cron-задачи для синхронизации данных Stripe будет временно размещен в репозитории вместе с остальным проектом. Перед обновлением инфраструктуры желательно проверить, что все работает, как ожидалось.

Как может выглядеть cron-задача, показано во фрагменте кода из текущего проекта (полный файл доступен на GitHub (https://oreil.ly/a6zxC)).

```
onModuleInit() {
  this.addCronJob('sync_stripe_orders', '5 00 * * *',
    this.cronSyncStripeOrders.bind(this));
}
…
async cronSyncStripeOrders(syncDate: Date) {
  this.logger.debug(`Синхронизация заказов Stripe начата в ${new Date()}`);

  const previousDate = syncDate.setDate(syncDate.getDate() - 1);

  // Установка параметров для получения инвойсов, созданных со вчерашнего дня
  const params = {
    created: {
      gte: previousDate,
    },
  };

  // Получение инвойсов из Stripe
  const { data: latestInvoices } = await this.stripe.invoices.list(params);

  // Проверка наличия новых инвойсов для обработки
  if (latestInvoices.length === 0) {
    this.logger.debug(`Нет новых инвойсов Stripe с ${previousDate}`);
    return;
  }

  // Цикл по инвойсам и создание новых записей в БД
  for (const invoice of latestInvoices) {
    const orderRecord = {
      name: invoice.account_name,
      stripeInvoiceId: invoice.id,
      total: invoice.total,
    };

    try {
      const doesOrderRecordExist = await this.prisma.order.findUnique({
        where: { stripeInvoiceId: invoice.id },
      });

      if (!doesOrderRecordExist) {
        await this.prisma.order.create({ data: orderRecord });
      }
    } catch (error) {
      this.logger.debug(`Произошла ошибка с инвойсом ${invoice.id}: ${error}`);
    }
  }
  this.logger.debug(`Синхронизация заказов Stripe завершена в ${new Date()}`);
}
```

Разберем ключевые моменты. Во-первых, cron-выражение `'5 00 * * *'` определяет расписание синхронизации. Экспериментировать с выражениями можно на сайте Cronitor (https://crontab.guru/#5_00_*). Это выражение запускает код

в 00:05 каждую ночь, что обеспечивает синхронизацию данных с максимальной информативностью и минимальным влиянием на пользователей. Учитывайте часовые пояса — их влияние на систему может оказаться неожиданным. Например, если сервер в поясе UTC-5, а пользователь в UTC-8, то со своевременной синхронизацией могут возникнуть проблемы. Полный код доступен в репозитории (https://oreil.ly/Bs9Hs).

Дополнительные замечания по cron-задачам

Джефф Грэм делится дополнительными наблюдениями о том, на что стоит обращать внимание при работе с cron-задачами.

> При росте трафика или нагрузки выполнение cron-задач часто сталкивается с проблемами: на них затрачивается все больше времени, что приводит к таймаутам, долгим запросам и сбоям. В худшем случае задача не завершает список операций, а перезапускается в течение дня снова и снова. Страшный сон!
>
> Для предотвращения подобных ситуаций критически важен мониторинг таких задач. Он позволит заранее обнаружить проблему и внести изменения. Стоит учесть, что все задачи по-своему уникальны: одни могут стабильно работать в исходном виде, а другие потребуют доработок и оптимизации.

Оповещения и мониторинг

В работе этой cron-задачи есть еще один почти незаметный аспект. Хотя логируются все ошибки, некоторые из них могут требовать более срочного реагирования. Здесь появляется необходимость в инструментах мониторинга. Вероятно, этим займется команда DevOps, так как она отвечает за управление ресурсами, а также за инциденты на уровне инфраструктуры. Однако если возникают программные проблемы, их решение и поиск первопричин ложатся на вашу команду разработчиков.

Следует отслеживать такие показатели, как количество ошибок, появившихся за определенный период; конкретные типы ошибок; другие параметры, которые вы согласуете с командой DevOps. Мониторинг позволяет активировать оповещения, помогая команде оперативно реагировать на проблемы. Обычно используются заранее определенные метрики, а также устанавливаются базовые показатели корректной работы приложения.

В большинстве случаев оповещения настраиваются на уровне инфраструктуры. Если задача неожиданно начинает запускаться слишком часто, команда DevOps может получать уведомления при достижении определенного порогового уровня использования ресурсов. Аналогично множество ошибок 403 в инструменте мониторинга может сигнализировать об атаке на систему.

Существуют и оповещения, необходимые команде разработчиков, например, о постоянном возникновении ошибок в коде. Более подробно мониторинг и оповещения рассматриваются в главе 12.

Благодаря сочетанию логирования, мониторинга и оповещений вы сможете оперативно классифицировать проблемы при их возникновении и получить данные для выявления первопричины.

Логирование

Следующий важный аспект — настройка системы логирования. При автоматическом выполнении задач периодически возникают ошибки. Отладка запланированных задач может быть сложной по нескольким причинам:

- недоступный или выдающий ошибки сервис;
- пропущенные записи;
- ошибки, возникающие только в определенное время суток;
- проблемы с базой данных.

Иногда ошибки невозможно воспроизвести, так как они возникают при специфических условиях. В таких случаях помогает анализ логов — подробнее об этом поговорим в главе 9. В коде предусмотрено множество строк логирования, поскольку именно в этих местах задачи могут возникать ошибки, приводящие к другим проблемам. При проектировании логирования учитывайте данные, которые понадобятся для отладки: идентификаторы, статусы, имена и другие параметры, которые можно сопоставить в разных частях системы.

Дополнительно рекомендуется использовать в логах уникальные поисковые метки. По мере роста продукта при каждом вызове эндпоинта генерируется огромное количество логов. Поэтому возможность быстро отфильтровать логи по конкретному событию критически важна для эффективной отладки.

Проблемы синхронизации данных и выполнения задач

Когда вы определите *первопричину* (источник ошибки в коде или конфигурации окружения) и придумаете, как ее исправить, вам потребуется найти способ перезапустить упавшее задание. Рекомендуемой практикой для бэкенда является реализация механизма повторных попыток для cron-заданий и фоновых задач. При большинстве проблем с синхронизацией данных устранить их позволяет повторная загрузка записей. В функции повторной попытки можно быстро подсчитать количество обновленных записей, чтобы оценить масштаб проблемы до ее исправления.

Я не буду подробно разбирать код фонового задания для отправки писем — его можно найти на GitHub (https://oreil.ly/oR5sA). Система оповещений будет полезна команде разработчиков при задачах, чувствительных ко времени и передающих

клиентам важную информацию. Она может включать автоматический ответ клиентам или разработчикам и одновременное уведомление службы поддержки о произошедшем. В перспективе метрики мониторинга смогут автоматически инициировать повторные попытки. В большинстве случаев для отладки проблем с заданиями потребуется тесное взаимодействие с командой DevOps из-за особенностей интеграции этих задач с инфраструктурой.

Перспективы развития

По мере роста числа пользователей будет увеличиваться количество задач и объем обрабатываемых ими данных, что приведет к большему количеству ошибок. Сейчас идеальный момент в проекте для реализации механизма остановки задач. Иногда задачи выполняются неожиданно, и вам нужен способ быстро их остановить. Вы можете реализовать механизмы остановки для каждой задачи или только для тех, которые сильнее всего влияют на систему в целом. Если вы изначально встроите эту функциональность в первую задачу, то при неизбежном копировании кода для создания новых задач она уже будет содержать этот инструмент.

При создании задач есть риск, что, пока они успешно выполняются, их код будет обновляться недостаточно часто. Чтобы избежать превращения кода задач в устаревший, всегда держите в бэклоге тикет на обновление пакетов и рефакторинг для этих задач. Достаточно выполнять такие обновления хотя бы раз в квартал, чтобы код оставался рабочим и актуальным.

Заключение

В этой главе мы научились создавать cron-задачу для синхронизации данных Stripe с корпоративной базой данных и фоновую задачу для отправки email-уведомлений пользователям о заказах. Здесь особенно важно глубокое понимание архитектуры, чтобы функциональность не терялась и не забывалась. Также мы увидели, насколько тесно должны взаимодействовать команда разработчиков и команда DevOps.

От вас не требуется быть DevOps-инженером, чтобы понимать основы работы инфраструктуры и использования настроенных инструментов. Может показаться, что вы должны уметь все, но это не так. Учитесь опираться на экспертные знания других специалистов — это поможет вашей команде работать эффективнее.

ГЛАВА 7

Тестирование бэкенда

На данном этапе проекта у вас уже накопилось значительное количество кода. В нем присутствуют сервисы, контроллеры и интеграции. Хотя вы с командой прилагаете все усилия, чтобы избежать *регрессионных ошибок* (когда ранее работающая функциональность ломается из-за несвязанных изменений), это неизбежно произойдет по мере роста кода, и это нормально. Именно поэтому для ключевых функций приложения необходимо писать тесты.

В бэкенде следует тестировать:

- возникновение ошибок в корректных сценариях;
- возврат данных в правильном формате;
- вызов методов с корректными параметрами.

Написание модульных тестов помогает предотвращать регрессионные ошибки, лучше понимать ожидаемое поведение кода и повышает его поддерживаемость благодаря написанию более лаконичного и модульного кода.

В этой главе мы рассмотрим:

- компромиссы между наличием тестов и их отсутствием;
- как писать тесты с использованием Jest;
- важность мок-данных (mock data).

Независимо от того, начинаете вы проект с нуля или наследуете существующий, тесты могут стать отличным подспорьем. Это также отличное время для совместной работы с продуктовой командой над текущим и будущим состоянием продукта. Написание тестов часто заставляет внимательнее посмотреть на то, что именно ожидается от той или иной функциональности.

Зачем тратить время на тесты

Разработчики часто расходятся во мнениях о том, как следует реализовывать тестирование. Здесь могут возникать жаркие споры, а значит, как и во всем остальном, придется искать компромиссы. Ключевой аспект — уровень покрытия тестами (test coverage) проекта. Во многих проектах достичь почти 100 % покрытия для бэкенда вполне реально, но разумнее ориентироваться на 90 % или выше. Наступает момент,

когда команде приходится решать, стоит ли писать дополнительные трудоемкие тесты, оценивая их пользу относительно затрачиваемого времени.

Написание тестов действительно замедляет циклы выпуска на начальном этапе. Тесты могут долго выполняться на этапе CI (Continuous Integration), отнимая время от разработки фич или рефакторинга. Однако преимущества весомее затрат, если соблюдать баланс. Да, вначале приходится жертвовать скоростью, но тесты предотвращают множество проблем и повышают уверенность команды в коде по мере роста приложения. Благодаря тестам разработчики приобретают новый опыт, так как могут вносить изменения, не сильно опасаясь регрессионных ошибок и неожиданных сбоев в других частях системы.

Тесты дают возможность продумать и задокументировать ожидаемое поведение кода. Они могут даже вызывать дополнительные вопросы по продукту, а иногда выявляют проблемы, которые заставляют пересмотреть архитектуру. Автоматизированные регрессионные тесты также помогают предотвратить попадание устранимых багов в продакшен.

Здесь полезно организовать совместную работу с командой QA, если такая команда есть. Если нет, то команде разработчиков придется частично взять на себя функции QA. В большинстве случаев разработчики сосредоточены на написании *модульных тестов*. Модульные тесты нужны, чтобы убедиться, что код реализован корректно и работает, как ожидается для различных сценариев использования. Команда QA может подключиться и написать автоматизированные сквозные тесты (end-to-end, e2e) для проверки вашей среды разработки. *Сквозные тесты* предназначены скорее для проверки сценариев взаимодействия пользователя (user flow), чем для тестирования кода. Для этой проверки хорошо подходят такие инструменты, как Postman (https://www.postman.com) и Thunder Client (https://www.thunderclient.com). Вы можете создавать наборы тестов для проверки среды разработки, чтобы убедиться, что эндпоинты действительно работают как ожидается.

По мере роста продукта и появления новых бизнес-правил становится критически важным иметь возможность проверять, что эндпоинты корректно обрабатывают данные. Некоторые проблемы при ручном тестировании могут остаться незамеченными, особенно при смене сотрудников. Наличие тест-кейсов как для модульных, так и для сквозных тестов — это форма документации, которая должна поддерживаться в актуальном состоянии.

Если новое изменение ломает тест, вы можете увидеть исходные предположения и перепроверить их актуальность с командой разработчиков или продуктовой командой. Стоит отметить, что написание тестов с такой степенью детализации может замедлить выпуск, так как необходимо учитывать время, необходимое для их написания и успешного прохождения. Поэтому важно закладывать это время в оценку тикетов.

Реализация кода и его развертывание без тестов могут быть полезными, когда нужно быстро добавить новую функцию или исправить ошибки. Но вы не можете быть

уверены, что при этом не сломали что-то, казалось бы, не связанное. Например, обновление схемы данных продуктов и переименование поля могут неожиданно нарушить функцию получения данных о заказах.

Может показаться, что тесты можно добавить позже, главное — выпустить следующую версию вовремя. Но это рискованное предположение. Лучше писать тесты параллельно с реализацией, чтобы не упустить ничего перед релизом. Кроме того, тесты часто откладываются в бэклог из-за появления новых ошибок и функционала.

Объясните продуктовой команде, почему написание тестов — залог более качественного продукта для пользователя. Покажите, как это снижает необходимость экстренных исправлений из-за неожиданных регрессий. Как только команда увидит ценность контроля качества, она начнет предлагать сценарии, которые помогут написать еще более качественные тест-кейсы.

Как подходить к написанию тестов

Хотя такие подходы, как разработка через тестирование (TDD, test-driven development) или разработка через поведение (BDD, behavior-driven development), могут быть полезны, их внедрять необязательно. Главное, четко понимать, для чего именно пишутся тесты. Этот важный аспект часто упускается в большинстве курсов и публикаций, посвященных тестированию. При написании модульных тестов для бэкенда вы проверяете конкретную функциональность.

Что именно должно происходить при запросе к эндпоинту? Возможно, потребуются тесты на потенциальные ошибки, которые могут вернуться в ответе. Также важно убедиться, что ожидаемые функции действительно вызываются, причем с правильными параметрами, и возвращают ожидаемые данные.

Анализируйте код построчно. Генерируется ли ошибка? Напишите для нее тест. Вызывается ли функция? Напишите тест, проверяющий корректность передаваемых параметров. Старайтесь соблюдать баланс между имитацией функций и тестированием реального кода. Если тестируется установленный модуль, разумно замокать его функциональность. Для внутреннего кода, написанного вашей командой, обычно допустимо тестировать реальные вызовы. Наличие таких тестов с самого начала сэкономит массу времени, обычно затрачиваемого на отладку при добавлении нового функционала и рефакторинге кода вами и другими разработчиками.

Благодаря тестированию сразу становится видно, почему какое-то новое изменение привело к сбою. Тесты также способствуют осмыслению того, как код должен работать по мере своего развития. В процессе обработки упавших тестов и анализа корректного поведения кода вы можете обнаружить участки, где логика поддается улучшению. Тесты должны фокусироваться на ключевых моментах: условиях в коде, возвращаемых HTTP-статусах (HTTP status codes), влиянии

прав пользователей (user permissions) на результаты API-запросов (API requests) и других факторах, изменяющих итоговый результат для пользователя или сервиса.

Рассмотрим метод `create` в `orders.controller.ts` и напишем для него модульные тесты:

```
@Post()
public async create(@Body() user: User, order: CreateOrderDto): Promise<Order> {
  if (!user) {
    throw new HttpException('Неавторизован', HttpStatus.UNAUTHORIZED);
  }
  if (!user.permissions.includes('create:orders')) {
    throw new HttpException('Запрещено', HttpStatus.FORBIDDEN);
  }
  if (!order) {
    throw new HttpException('Нет данных заказа', HttpStatus.BAD_REQUEST);
  }
  if (order.products.length === 0) {
    throw new HttpException('Нет товаров в заказе', HttpStatus.CONFLICT);
  }
  if (!order.total) {
    throw new HttpException('Нет итоговой суммы', HttpStatus.CONFLICT);
  }
  try {
    const newOrder = await this.ordersService.createOrder(order);
    return newOrder;
  } catch (err) {
    throw new HttpException('Произошла ошибка', HttpStatus.NOT_FOUND);
  }
}
```

Здесь видны возможные ошибки, возникающие до вызова сервисного метода. При выполнении любого из условий клиенту вернется соответствующая ошибка. Сообщения об ошибках могут быть как конкретными, так и общими, главное, чтобы они не раскрывали персональные данные или иную информацию, которую могут использовать злоумышленники.

Рекомендуется делать сообщения короткими и отправлять соответствующий статус-код. Теперь рассмотрим этот метод с точки зрения тестирования, разбирая его построчно. Читая код, можно выделить пять тест-кейсов для выдаваемых ошибок, один тест-кейс для успешного вызова сервиса и один тест-кейс для ошибки, возникающей в сервисе. Давайте разберем, как написать эти семь тестов с использованием Jest:

```
import { Test, TestingModule } from '@nestjs/testing';
import { OrdersV1Controller } from './orders.controller';
import { OrdersService } from './orders.service';

const user = {
  id: 1001,
  email: 'tester@rest.com',
```

```
  name: 'Tester Rest',
  permissions: ['get:orders', 'create:orders'],
};

const order = {
  name: 'Biggest order',
  total: 125.99,
  stripeInvoiceId: 'stripeInvoiceId',
};

describe('OrdersController', () => {
  let controller: OrdersV1Controller;
  let ordersService: OrdersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      controllers: [OrdersV1Controller],
      providers: [OrdersService],
    }).compile();

    controller = module.get<OrdersV1Controller>(OrdersV1Controller);
    ordersService = module.get<OrdersService>(OrdersService);
  });

  it('выдает ошибку "не авторизован", если пользователь не задан', async () => {
    await controller.create(undefined, order);

    expect(controller.create).toThrowError('Unauthorized');
  });

  it('выдает ошибку "доступ запрещен", если у пользователя недостаточно прав',
async () => {
    const badPermissionsUser = {
      id: 1001,
      email: 'tester@rest.com',
      name: 'Tester Rest',
      permissions: ['get:products'],
    };

    await controller.create(badPermissionsUser, order);

    expect(controller.create).toThrowError('Forbidden');
  });

  it('выдает ошибку в запросе, если заказ не задан', async () => {
    await controller.create(user, undefined);

    expect(controller.create).toThrowError('No order data');
  });

  it('выдает ошибку соответствия, если товаров нет в заказе', async () => {
    const orderWithoutName = {
      total: 125.99,
```

```
      stripeInvoiceId: 'stripeInvoiceId',
    };

    await controller.create(user, orderWithoutName);

    expect(controller.create).toThrowError('No order name');
  });

  it('выдает ошибку соответствия, если отсутствует сумма заказа', async () => {
    const orderWithoutTotal = {
      name: 'Biggest order',
      stripeInvoiceId: 'stripeInvoiceId',
    };

    await controller.create(user, orderWithoutTotal);

    expect(controller.create).toThrowError('No order total');
  });

  it('успешно создает новый заказ', async () => {
    controller.create(user, order);
    const newOrder = await ordersService.createOrder(order);

    expect(ordersService.createOrder).toBeCalledWith(order);
    expect(ordersService.createOrder).toReturnWith(newOrder);
  });

  it('выдает ошибку "не найдено", если сервис не работает', async () => {
    try {
      controller.create(user, order);
      await ordersService.createOrder(undefined);
      expect(ordersService.createOrder).toThrowError();
    } catch (e) {
      expect(e.message).toBe('Something happened');
    }
  });
});
```

Стоит упомянуть популярные ИИ-инструменты, которые могут помочь в написании тестов, такие как Copilot (https://oreil.ly/s6glK) или ChatGPT (https://openai.com/chatgpt). Подобные инструменты ускоряют процесс создания тестов и генерации сценариев.

Все эти тесты основаны на сценариях, описанных в коде контроллера. При написании тестов в целом обычно не нужно беспокоиться о проверке реализации вспомогательных функций и сервисов — для них существуют отдельные модульные тесты. В этом одно из преимуществ модульного подхода: он делает написание тестов более прозрачным благодаря четкому разделению ответственности.

Для каждого сценария предусмотрены тестовые данные и ожидаемый ответ. Такой подход гарантирует покрытие основных случаев. Хотя при обсуждении

с продуктовой командой и QA важно стараться предугадать граничные случаи, их можно добавлять в тесты и позже, по мере обнаружения.

При написании тестов мне очень помогает одновременное открытие файла с тестами и файла с кодом. Это позволяет проверять, протестирована ли каждая строка кода по мере продвижения по файлу.

Существует несколько подходов к сквозному тестированию API. Можно использовать инструмент вроде Postman, поскольку он удобнее для неразработчиков, или воспользоваться такими инструментами, как Cypress (https://www.cypress.io) или Playwright (https://playwright.dev). На рис. 7.1 показано, как выглядят тесты в Postman. Вы можете импортировать JSON-файл с этими тестами (https://oreil.ly/Eerfp) в Postman и изучить тесты самостоятельно. В главе 24 мы рассмотрим запуск сквозных тестов в CI/CD-пайплайне.

Рис. 7.1. Сквозные тесты в Postman

Термин *«сквозное тестирование»* (e2e) неоднозначен — его трактовка зависит от того, с кем вы об этом говорите. Некоторые утверждают, что сквозными тестами называют только такие, которые охватывают и фронтенд и бэкенд вместе. Другие считают, что сквозные тесты бывают и фронтендными, и бэкендными, главное, чтобы они фокусировались на функциональности, а не на реализации кода, как модульные. Важно понимать, что не существует единственно верного определения сквозных тестов. Различие в трактовках может зависеть от конкретной организации.

Если вы решите использовать программный подход с инструментом вроде Cypress, тесты будут выглядеть примерно так:

```
it('/v1/orders (POST)', () => {
    return request(app.getHttpServer())
      .post('/v1/orders')
      .expect(200)
      .expect(order);
    });
```

Таким образом, синтаксис схож с тем, что используется для модульных тестов. При программном подходе важно учитывать время, которое потребуется команде для освоения конкретной библиотеки. Однако сценарии тестирования как для модульных, так и для сквозных тестов будут очень похожи. Модульные тесты проверяют реализацию кода и манипуляции с данными, тогда как сквозные тесты анализируют поток данных на более высоком уровне — аналогично тому, как это делает фронтенд или другой сервис. Подробнее о сквозном тестировании вы узнаете в главе 24.

Мок-данные

Вы могли заметить, что в некоторых тестовых случаях использовались мок-данные. Эти данные должны учитывать различные сценарии, которые могут возникнуть в проекте. Для порядка в документации полезно хранить моки в отдельных файлах с четкими названиями, хотя иногда разница между тестовыми случаями сводится к одному полю. В таких ситуациях может потребоваться обновление соглашений по коду. Главное, чтобы команда договорилась о наилучшем подходе для конкретного сценария.

Мок-данные можно получить напрямую из базы или с помощью инструментов вроде Faker (https://fakerjs.dev/guide) или Falso (https://github.com/ngneat/falso). На этом этапе также целесообразно повторно проверить типы данных. При их создании согласовывайте с продуктовой командой ожидаемое поведение в различных сценариях. Допускается делать предположения по поведению и типам, но обязательно убедитесь, что они реально соответствуют требованиям к продукту.

Еще один вариант работы с мок-данными — создание тестовых данных непосредственно в БД. Это обеспечивает доступность моков для сквозных тестов, что особенно полезно для демонстраций продуктовой команде и стейкхолдерам: можно протестировать полный сценарий без модификации кода. Кроме того, это позволяет им самостоятельно проверить функциональность и выявить необходимые изменения.

Хороший способ начать мокать данные — создать полный объект для каждой модели. Например, вы создали мок-данные пользователя следующим образом:

```
const user = {
  id: 1001,
  email: 'tester@rest.com',
  name: 'Tester Rest',
  permissions: ['get:orders', 'create:orders'],
};
```

Здесь есть все необходимое согласно схеме пользователя. Если потребуется изменить поля, можно просто обновить этот объект в тесте для конкретного случая. Пример такого подхода показан в тест-кейсе проверки прав пользователя, где модифицируется поле `permissions`.

Еще один эффективный способ создания мок-данных — использование начальных данных из вашей БД. Здесь важно отметить, что эти данные должны обновляться синхронно с изменениями схемы. Это гарантирует, что при развертывании проекта на новом компьютере не возникнет проблем с отсутствующими данными или несовместимыми типами.

Альтернативный взгляд на мокирование данных

Вот еще один ценный совет от Итана Брауна.

Я считаю, что наличие хорошей инфраструктуры для мок-данных — недооцененный принцип в разработке программного обеспечения. Большинство реальных приложений работают с очень специфичными типами данных, и разработчики постоянно тратят время на изобретение тестовых данных для каждой мелочи. Наличие хороших и реалистичных фабрик тестовых данных, особенно если встроить их в сам процесс создания классов/типов, может кардинально повысить продуктивность.

Например, если разработчик начинает работать над функциональностью, связанной с объектом «пользователь», я не хочу, чтобы он:

1) создавал мини-объект пользователя, в котором легко упустить что-то важное и который может сломаться в продакшене;
2) тратил много времени на ручное создание мок-объектов.

Лучше, чтобы он просто вызвал функцию `FakeUserFactory.create(10)` и получил 10 пользователей, как если бы они были созданы в реальных условиях. Это также полезно для демонстраций продукта, пробных аккаунтов или симуляций.

Также хорошо иметь набор тестовых данных, охватывающих различные реальные сценарии. Даже если это просто файл с 10–15 фиктивными пользователями, из которых разработчики выберут подходящие для тестирования, эти данные могут служить напоминанием о том, как следует обрабатывать «грязные» реальные данные.

Для файла со списком пользователей я бы включил таких, у которых:

- супердлинное имя («Сэр Чарльз Ксавьер Тестер Тесталот III»);
- однословное имя («Принц»);
- имя в расширенной кодировке («Dvořák»).

Когда junior-разработчик просматривает такой список, даже если он просто ищет тестовые данные, он может подумать: «О черт, у меня код сбойнет, если не будет имени или фамилии».

Заключение

В этой главе мы рассмотрели тестирование бэкенда и его важность. Вам будут встречаться разработчики, которые раньше не писали столь детализированных тестов, и они могут сопротивляться. Однако учитывайте: каждый раз, когда в продакшене находят баг, попавший туда после очередного развертывания, в компании начинается мини-паника и возникает срочная необходимость сделать хот-фиксы. Тесты помогают этого избежать, даже если они требуют дополнительного времени разработчиков. В главах 24 и 25 мы поговорим о том, как разворачивать и интегрировать тесты в этот процесс.

Также мы разобрали несколько разных способов тестирования кода. Модульные тесты проверяют реализацию кода, а сквозные позволяют удостовериться, что последовательность действий выполняется должным образом, если смотреть с позиции пользователя. Также можно сказать, что любое тестирование, выполняемое продуктовой командой, включает в себя проверку приемки пользователя, потому что продакты отвечают за функциональность сервиса с точки зрения пользователя. Тестирование — это групповая работа, поэтому полезно обсуждать ее со всеми заинтересованными сторонами, чтобы чувствовать уверенность: все сделано как надо.

ГЛАВА 8

Вопросы безопасности бэкенда

Безопасность вашего приложения будет одним из главных приоритетов для команды разработчиков, продуктовой команды и всей организации. Нельзя допустить, чтобы неавторизованные субъекты получили доступ к данным или функционалу. Соответственно, необходимо рассматривать безопасность приложения со всех сторон — аутентификации (authentication), авторизации (authorization), валидации, распространенных типов атак и внешних зависимостей.

Безопасность — это обширная тема, которой посвящены целые книги. В этой главе мы сосредоточимся конкретно на мерах для бэкенда, хотя охват вопросов безопасности гораздо шире. Если вам повезет работать со службой информационной безопасности, или ИБ (Security team), то она будет проверять всю техническую инфраструктуру компании вплоть до приложений, которые разрешается устанавливать на рабочий ноутбук.

В этой главе вы узнаете о:

- методах аутентификации и случаях их применения;
- авторизации пользователей для предоставления различных уровней доступа к функциям и данным в приложении;
- причинах, по которым обычно стоит выбирать готовые (out-of-the-box) решения;
- разделах OWASP Top 10 (Open Web Application Security Project), относящихся к бэкенду;
- лучших практиках безопасности, нормативных требованиях и источниках для углубленного изучения различных аспектов безопасности.

В этой главе я лишь поверхностно затрагиваю заявленную тему и оставлю вам множество ресурсов для изучения подробностей, но покажу, как реализовать некоторые хорошие практические методы обеспечения безопасности. На их основе вы сможете расширить защиту в соответствии с нормативными требованиями, применимыми к вашему продукту, такими как:

- HIPAA (Health Insurance Portability and Accountability Act) (https://oreil.ly/uLXpC) для медицинских приложений;
- PCI DSS (https://oreil.ly/nALD1) для финтех-приложений;

- GDPR (General Data Protection Regulation) (https://gdpr.eu/checklist) для практически любых приложений.

Давайте начнем с того, как пользователи входят в приложение, используя аутентификацию.

Аутентификация

Аутентификация — это процесс проверки подлинности пользователя в приложении. Она осуществляется через определенный метод входа. Среди распространенных методов аутентификации (https://oreil.ly/65NI_), с которыми вы столкнетесь:

- аутентификация по паролю;
- многофакторная аутентификация (MFA, multifactor authentication);
- аутентификация по токену;
- биометрическая аутентификация;
- открытая авторизация (OAuth, open authorization);
- одноразовый пароль (OTP, one-time password).

В зависимости от приложения вы можете комбинировать эти методы, чтобы предоставить пользователям больше вариантов входа. Как правило, в любом приложении используется хотя бы один или два из них.

Подтверждение личности пользователя — сложная задача. Злоумышленники могут пытаться выдать себя за реальных пользователей с помощью атак методом перебора (brute force), фишинга или перехвата токена сессии. Ваша цель — не допустить несанкционированный доступ.

Наиболее распространенный метод аутентификации, который вам предстоит реализовывать, — это вход по паролю, имеющий различные формы. Пароль пользователя будет безопасно храниться в вашей базе данных, и только комбинация имени пользователя и пароля позволит ему войти в систему. Более глубокие аспекты безопасности включают такие темы, как хеширование паролей с солью (salted password hashing) (https://oreil.ly/JKRfu), шифрование (https://oreil.ly/TVMX3) и расшифрование конфиденциальных персональных данных пользователя, таких как номера социального страхования (аналог СНИЛС) или данные платежных карт. Эти темы выходят за рамки главы, но я рекомендую вам изучить их дополнительно, чтобы лучше понимать, как и когда они применяются.

Аутентификацию по паролю можно сделать еще более безопасной, добавив требования к паролю, такие как минимальная длина и использование специальных символов. Однако основной риск этого метода связан с поведением пользователя. Если он использует один и тот же пароль в нескольких системах, вы мало что сможете сделать, полагаясь только на метод аутентификации по логину и паролю.

Именно здесь на помощь приходят такие методы, как MFA (многофакторная аутентификация) (https://oreil.ly/vS_7a) и OTP (одноразовый пароль) (https://oreil.ly/xYoyP), поскольку они добавляют дополнительный уровень идентификации поверх пароля. Для этого пользователю потребуется доступ к другим инструментам, таким как электронная почта, номер телефона или специальное приложение, например Google Authenticator (https://oreil.ly/LpaC_) или Okta Verify (https://oreil.ly/wff9u). Обычно пользователь получает код, который необходимо ввести в приложении для завершения процесса аутентификации. Этот дополнительный шаг предотвращает вход злоумышленников, даже если они располагают паролем.

Мы уже кратко упоминали в главе 5 некоторые сервисы для аутентификации, но стоит повторить их здесь: Auth0 (https://auth0.com/docs), FusionAuth (https://fusionauth.io/docs), Amazon Cognito (https://oreil.ly/06RmU), SuperTokens (https://supertokens.com/docs/guides) и Clerk (https://clerk.com). Также есть решения с открытым исходным кодом, такие как Passport.js (https://www.passportjs.org), NextAuth.js (https://next-auth.js.org) и встроенная функциональность в NestJS (https://oreil.ly/FIRPN). Как правило, стоит выбрать сторонний сервис для аутентификации, если требуется повышенный уровень безопасности из-за отраслевых регуляторных требований.

Вам также предстоит углубиться в детали того, как хранить токены для пользовательской сессии и что делать, если их сроки истекли. В процессе этого вы познакомитесь с такими темами, как поток кода авторизации (authorization code flow) (https://oreil.ly/UCB28) и механизм Proof Key for Code Exchange (PKCE) (https://oreil.ly/Kdzeg). Или, возможно, вы решите, что лучше разлогинивать пользователя и требовать повторный вход через определенное время.

При оценке вариантов важно учитывать пользовательский опыт. Да, биометрическая аутентификация безопаснее, но будут ли пользователи ею пользоваться? Вам предстоит найти баланс между удобством пользователей и методами аутентификации, обсудив это со своей командой и командой ИБ.

После успешной аутентификации пользователя необходимо определить уровень его доступа к системе. Не всем пользователям требуется одинаковый функционал — он зависит от их роли в приложении.

Авторизация

Даже если пользователь может войти в систему, это не означает, что ему должен быть доступен весь функционал. Здесь начинает работать *авторизация* — процесс назначения прав доступа после аутентификации. Именно авторизация определяет различия между ролями пользователей. Например, сотрудник службы поддержки получает доступ ко множеству аккаунтов, тогда как обычный клиент — только к своему.

Авторизация — одна из причин, почему важна надежная аутентификация. Компрометация аккаунта обычного пользователя — это плохо, но взлом учетной записи

администратора — *катастрофа*. Далее предстоит выбрать модель контроля доступа. Базовый принцип, который стоит реализовать в авторизации, — принцип минимальных привилегий (https://oreil.ly/HNLFe).

Доступ по принципу минимальных привилегий (least-privilege access) гарантирует, что пользователи получают только те права, которые действительно необходимы для выполнения задач в рамках их уровня доступа. Например, сотруднику, управляющему документацией приложения, не требуются такие же права, как сотруднику, занимающемуся онбордингом пользователей. Этот принцип должен лежать в основе любой другой реализуемой вами системы контроля доступа.

Наиболее распространенным подходом является ролевая модель контроля доступа (RBAC, role-based access control) (https://oreil.ly/PfS9v). RBAC позволяет назначать пользователям различные роли в зависимости от уровня доступа, который эти роли предоставляют. Многие сервисы аутентификации включают встроенную поддержку RBAC. Эту функциональность можно реализовать и внутри системы, но следует учитывать, что это, вероятно, потребует разработки фронтенд-компонентов, позволяющих внутренним пользователям управлять ролями и учетными записями пользователей.

Вот небольшой пример реализации RBAC, который можно использовать для создания собственных функций проверки ролей пользователей и предоставления доступа к различным частям приложения:

```
export const rbacConfig = {
  rolesConfig: [
    {
      roles: ['Customer'],
      permissions: ['get:order', 'create:order', 'get:product'],
    },
    {
      roles: ['Support'],
      permissions: ['get:order', 'update:order', 'delete:order', 'get:product'],
    },
    {
      roles: ['Store'],
      permissions: [
            'get:product',
                'delete:product',
                'create:product',
                'update:product'
          ],
    },
  ],
};

const canCreateOrder = (user: User) => {
  if (user.roles.includes(Roles.Customer)) return true;
  return false;
};
```

Также существует модель управления доступом на основе атрибутов (ABAC, Attribute-Based Access Control) (https://oreil.ly/MTLH_). ABAC обеспечивает предельно точный контроль действий пользователей в зависимости от их ролей, ресурсов, к которым они обращаются, выполняемых операций и окружения. Это гибкий подход к безопасности, позволяющий точно настраивать доступ под конкретные задачи пользователя. Существующие роли можно расширять или использовать как шаблоны для новых. ABAC применяется, например, в AWS IAM (Identity and Access Management) — это их стандартный механизм.

Другой взгляд на ABAC и RBAC

Итан Браун делится ценными наблюдениями о реализации различных типов авторизации. Вот некоторые из его мыслей.

> На мой взгляд, и у RBAC, и у ABAC есть различные подводные камни, которые могут поставить безопасность под угрозу, и я не стал бы называть ABAC изначально более безопасным. Основная опасность ABAC заключается в том, что при столь детализированном контроле доступа роли могут содержать десятки или сотни правил, что затрудняет анализ и проверку конкретных сценариев авторизации. Зачастую обширные списки политик ABAC копируются из одной роли в другую просто для экономии времени, но при этом легко пропустить политики, которые были включены случайно и не должны были там оказаться.
>
> RBAC в этом отношении лучше — в более простой системе авторизации меньше мест, где могут скрываться уязвимости безопасности, — но RBAC не будет одинаково надежным для всех систем. Преимущество ABAC в его гибкости: он позволяет создавать новые роли и шаблоны доступа на будущее, даже те, которые вы пока не предусмотрели. Если вы предполагаете, что в приложении будут появляться все более сложные правила авторизации, комбинация ABAC и RBAC (я редко встречаю ABAC без RBAC) может стать более удачным вариантом.

Основной недостаток ABAC заключается в сложности настройки таких систем. Ключевой аспект — корректное назначение атрибутов соответствующим ролям. Инициализация требует значительных усилий, но со временем окупается, если ваше приложение имеет сложные требования к правам доступа. По мере изменения ролей вы сможете копировать множество параметров для новых атрибутов. Однако при работе с ABAC будьте осторожны с копированием ролей: так вы можете случайно предоставить роли избыточные разрешения.

Вот небольшой пример определения ABAC:

```
export const abacConfig = {
  attributesConfig: [
```

```
    {
      action: 'CreateOrder',
      attributes: {
        user: {
          type: ['customer'],
        },
        resource: {
          type: 'order',
        },
      },
    },
    {
      action: 'GetOrder',
      attributes: {
        user: {
          type: ['customer', 'support'],
        },
        resource: {
          type: 'order',
        },
      },
    },
    {
      action: 'GetProduct',
      attributes: {
        user: {
          type: ['customer', 'support', 'store'],
        },
        resource: {
          type: 'product',
        },
      },
    },
  ],
};

const canGetOrder = ({ user, resource, action }: GetOrderProps) => {
  if (acceptedOrderUserTypes.includes(user.type) &&
      resource === 'order' &&
      action === 'GetOrder') {
    return true;
  }
  return false;
};
```

Последняя модель, которую я рассмотрю, — это модель управления доступом на основе политик (PBAC, policy-based access control) (https://oreil.ly/Hrl2R), очень похожая на ABAC. Обе модели относятся к *детализированному управлению доступом* (FGAC, fine-grained access control). Основное различие между ними заключается в том, что PBAC использует определенные вами политики, тогда как ABAC опирается на атрибуты. Это просто разные способы определения

ограничений доступа. PBAC также может быть доступна у вашего облачного провайдера.

Часто команда ИБ или команда поддержки пользователей приступает к управлению доступом после того, как разработчики и продуктовая команда задокументируют правила взаимодействия с приложением для различных пользователей. В других случаях вы сможете напрямую добавлять новые атрибуты или роли, а также изменять существующие разрешения. Хорошая практика — предоставлять всем пользователям минимально необходимые привилегии для начала работы, а затем расширять их по мере необходимости.

У всех этих моделей контроля доступа есть свои плюсы и минусы. RBAC, вероятно, реализуется быстрее всего, но со временем управление всеми ролями может стать запутанным. Если вы предполагаете, что приложение вырастет до уровня, где RBAC станет громоздким, стоит рассмотреть добавление ABAC или PBAC. Эти модели потребуют больше времени для настройки, но в долгосрочной перспективе ими будет проще управлять. При выборе модели задайте себе следующие вопросы:

- Какие типы пользователей имеются и какой уровень доступа нужен для каждого типа?
- Предоставляют ли используемые вами сервисы функциональные возможности контроля доступа?
- Какой уровень детализации (granularity) требуется для контроля доступа?
- Насколько гибкими или динамичными должны быть уровни доступа?
- Какие данные у вас есть для определения уровня доступа пользователя?

Это лишь поверхностное знакомство с темой авторизации. Рекомендую изучить дополнительные материалы из этого раздела. Вы можете выбрать одну из предложенных стратегий и совместно с командами ИБ и поддержки пользователей реализовать ее в масштабах компании.

Теперь, с пониманием основ аутентификации и авторизации, можно перейти к изучению наиболее распространенных угроз в интернете и того, как ваши действия помогают от них защититься.

OWASP Top 10

Независимо от того, знакомы ли вы с OWASP Top 10 (https://oreil.ly/4QqBE), следует детально изучить все пункты этого списка, чтобы понять аспекты безопасности вашего кода. Некоторые из них особенно актуальны для темы этой главы. Нарушение системы контроля доступа (broken access control) занимает первое место в списке уязвимостей, что указывает на важность этой темы.

Нарушение системы контроля доступа (https://oreil.ly/1tkhr) часто остается незамеченным при изменении ролей и пользователей. Типичная причина

проблемы — несоблюдение принципа минимальных привилегий. Ситуация может дойти до того, что вы или сотрудник команды безопасности будете постоянно обновлять права, а затем для упрощения выдадите всем одинаковые разрешения. Это приводит к тому, что пользователи, которым разрешено только чтение данных, получают возможность их записи, даже если они об этом не знают.

Еще один распространенный способ обхода системы контроля доступа — отсутствие такого контроля для определенных ресурсов. Вы можете не забыть добавить контроль доступа к POST-эндпоинтам, но случайно пропустить PUT или DELETE. Это позволяет пользователям обходить предполагаемые ограничения прав. Чтобы избежать подобного, следует по умолчанию запрещать доступ. Соответственно, доступ к ресурсам нужно предоставлять явным образом.

Еще одна проблема из Top 10 — инъекции (https://oreil.ly/-eyJe) из-за отсутствия валидации входных данных. Необходимо проверять все данные, поступающие с фронтенда или напрямую через эндпоинты. Некоторые разработчики ошибочно полагаются исключительно на фронтенд-валидацию, что неверно. При получении запроса с пользовательскими параметрами их значения должны валидироваться и очищаться для предотвращения SQL-инъекций и межсайтового скриптинга (XSS).

Ошибки в аутентификации (https://oreil.ly/1Ctrq) — еще одна уязвимость Top 10 — возникает из-за уязвимостей аутентификации: разрешение простых паролей, использование учетных данных прямо в URL или даже ненадежные механизмы сброса пароля. Часто такие уязвимости вызваны стремлением повысить удобство для пользователей. Это один из компромиссов, которые приходится учитывать при выборе типа аутентификации.

Чтобы избежать ошибок в аутентификации, можно применять следующие меры: требовать от пользователей сложных паролей, добавлять многофакторную аутентификацию и ограничивать количество попыток входа. Эти методы не являются абсолютно надежными — целеустремленный злоумышленник все равно способен обойти систему аутентификации. Однако комплексное использование различных уровней защиты может значительно снизить вероятность успешных атак.

Часто упускают из виду уязвимости, связанные с устаревшими компонентами (https://oreil.ly/oN47F). После обнаружения уязвимостей и выпуска соответствующих патчей в старых версиях пакетов или инструментов остаются хорошо документированные векторы атак. Своевременное обновление зависимостей критически важно как для безопасности, так и для поддержки кода. Еще один способ повысить защищенность — следить, чтобы инфраструктура использовала самые актуальные версии сервисов.

Последняя уязвимость из Топ-10, которую мы рассмотрим, поскольку она представляет особый интерес, — это небезопасная архитектура приложения (insecure design) (https://oreil.ly/7b8xU). Иногда архитектура приложения может быть небезопасна сама по себе или из-за недостаточной ее проработки. Для решения этой

проблемы целесообразно провести моделирование угроз (threat modeling) (https://oreil.ly/q8XOv). Этот процесс состоит из четырех шагов, где вам предстоит найти ответы на вопросы:

1. Над каким функционалом мы работаем?
2. Какие проблемы могут при этом возникнуть?
3. Как мы будем их решать?
4. Достаточно ли хорошо мы решили выявленные проблемы?

Типичный сценарий, где это может произойти, — оформление заказов. Существует ли способ, позволяющий пользователям обойти ограничение по количеству товаров в наличии? Могут ли они изменить детали заказа по ссылке из электронного письма? Есть ли вероятность, что пользователи увидят некорректную информацию из-за условных проверок в вашем коде?

Такие уязвимости сложнее обнаружить, поскольку иногда они являются частью требований к функционалу. В этом случае необходимо рассмотреть его со всех возможных сторон. Если для понимания функциональности требуется привлечь другие команды — организуйте созвон. Указывайте на любые проблемы безопасности, которые вы заметили, когда владелец продукта объясняет желаемую реализацию задачи. Это может полностью изменить подход к ее разработке.

Обязательно изучите оставшиеся уязвимости из Топ-10. Некоторые из них будут рассмотрены далее в главе 19, посвященной безопасности фронтенда. Однако существуют и общие практики безопасности, которые можно добавить в свой инструментарий, не относящиеся к конкретной теме.

Прочие важные практики

Независимо от отраслевых регуляторных требований, применяемых к вашему продукту, всегда следует реализовывать систему аудит-логов для отслеживания изменений данных или триггеров событий. Как минимум, включайте поля с датой обновления и ID пользователя, внесшего изменения. Это поможет выявить пользователей и сервисы, вызывающие проблемы (например, несанкционированный доступ к данным), а также определить момент начала атаки. Для отслеживания доступа к данным во времени можно также добавить аудит-логи для GET-запросов.

Использование инструментов вроде Snyk (https://snyk.io) или Veracode (https://www.veracode.com) позволяет автоматически сканировать код на наличие большинства известных уязвимостей безопасности. Такие инструменты интегрируются в DevOps-пайплайны — об этом пойдет речь в главе 27. Они также выявляют уязвимости в зависимостях. Получая отчеты об уязвимостях приложения, выделите время на их анализ и выбор оптимального способа устранения. Возможные варианты — замена библиотек или использование альтернативных сервисов.

Важно понимать, что существует несколько типов проверки безопасности. Первым следует внедрить статическое тестирование безопасности приложений (Static Application Security Testing, SAST) (https://oreil.ly/USD_5). Оно анализирует код на известные уязвимости и небезопасные паттерны, позволяя проверять код до его выполнения. Для проверки кода во время работы (аналогично действиям злоумышленника) применяется динамическое тестирование безопасности (Dynamic Application Security Testing, DAST) (https://oreil.ly/UeAJ-). Интеграция этих методов в пайплайн выполняется совместно с командой DevOps.

Рекомендуйте регулярное проведение пентестов (https://oreil.ly/m2X7l). Обычно для этого привлекают команды этичных хакеров, которые выявляют слабые места системы и предоставляют отчеты с находками и вариантами решений. Эти задачи могут также входить в зону ответственности команды ИБ. Некоторые компании запускают программы баг-баунти (bug bounties) — вознаграждений за найденные в рабочих приложениях уязвимости (https://oreil.ly/3QWN2). Чем больше векторов атаки будет протестировано, тем устойчивее станет продукт к реальным атакам.

Реальные атаки обычно исходят от «черных» хакеров (black-hat, https://oreil.ly/3QWN2). Они являются противоположностью белых (white-hat), то есть этичных хакеров, которые используют свои навыки только для выявления и устранения уязвимостей. «Черные» хакеры — это те, кому вы пытаетесь запретить несанкционированный доступ к различным частям вашего приложения. Существуют также «серые» (grey-hat) хакеры (https://oreil.ly/6_65O), которые обычно находят уязвимости без разрешения, но не используют их для причинения вреда.

Регулярно проводите отраслевые аудиты, чтобы убедиться, что приложение по-прежнему соответствует всем необходимым регуляторным требованиям. Вероятно, за этим будут следить юристы, но если такой команды нет, проверяйте код на соответствие закону. Например, обязательно надо следить, шифруются ли персональные данные пользователей и корректно ли предоставляется им доступ к данным.

Обновляйте зависимости своих компонентов как можно чаще, чтобы не пропустить исправления уязвимостей. Инструмент вроде Dependabot (https://oreil.ly/LBdN5) поможет вам в этом. Вы можете обнаружить, что некоторые приложения отстают на несколько крупных версий по определенным пакетам, даже несмотря на то что уязвимости хорошо известны и задокументированы. Это один из видов технического долга (tech debt), который стоит обсудить с продуктовой командой для приоритизации в бэклоге. Гораздо проще выполнять обновления постепенно, чем пытаться обновить все сразу после нескольких критических изменений.

Еще один аспект, который необходимо учитывать при работе с зависимостями, — это безопасность цепочки поставок (supply chain security). Мы часто используем открытое программное обеспечение, которое, в свою очередь, имеет собственные потенциально уязвимые зависимости. Эти уязвимости могут быть внесены преднамеренно, например, если пакет инициирует распределенную атаку типа «отказ в обслуживании» (DDoS) на ваш сервер (https://oreil.ly/S1Bld). Или они могут быть

случайными, например, если кто-то из команды установит поддельную версию пакета (https://oreil.ly/-rKKg). Для мониторинга уязвимостей, связанных со сторонними зависимостями в вашем приложении, можно использовать этот фреймворк безопасности цепочки поставок (https://oreil.ly/gzXlO).

Также необходимо регулярно обновлять токены доступа и учетные данные, используемые для работы со сторонними сервисами или инструментами. Я имела дело с компаниями, где одни и те же токены использовались годами, а затем приходилось тратить дни на выяснение причин сбоя системы из-за окончания срока действия токена. Обновляйте их как минимум несколько раз в год, но уточните у команды ИБ, есть ли у них конкретные регламенты для этой процедуры.

Учет опыта разработчиков при тестировании авторизации

Итан Браун делится ценными наблюдениями о работе над реальными проектами.

> При работе с авторизацией часто упускают из виду то, как ее реализуют разработчики. В одном крупном проекте со сложными требованиями к авторизации разработчики изначально назначили себе политики типа «суперпользователь» для ускорения реализации функционала. Однако при тестировании с реальными пользователями, имеющими ограниченные права, многие функции оказались неработоспособными — разработчики не тестировали сценарии от лица разных ролей, а действовали исключительно как суперпользователи. Это побудило нас внедрить политику «Разработчик», которая не давала повышенных прав напрямую, но позволяла переключаться между ролями через удобный «переключатель ролей».
>
> Это упростило процесс разработки: возможность смены ролей без перелогинивания, множественных аккаунтов или браузеров ускорила работу над ролевыми функциями. Подход не только снизил нагрузку на разработчиков, но и ненавязчиво напомнил разработчикам о необходимости учитывать ролевой доступ при реализации механизмов авторизации.

Выделите время, чтобы научиться выполнять базовые атаки самостоятельно! Эта практика относится к этичному хакингу (https://oreil.ly/2oF2I), где вы можете использовать навыки взлома для поиска и документирования уязвимостей, чтобы затем их устранить. Академия веб-безопасности от компании PortSwigger (https://oreil.ly/ZzNSw) предлагает полезные материалы по различным часто встречающимся типам атак. Вы также можете изучить инструменты, которые обычно используют злоумышленники, такие как sqlmap (https://sqlmap.org), Wireshark (https://www.wireshark.org), John the Ripper (https://github.com/openwall/john) или даже всю ОС Kali Linux (https://www.kali.org). От вас не требуется стать хакером, но вам будет полезно понимать, как все это работает с точки зрения злоумышленника. Вы сможете лучше защитить свой код.

Заключение

Эта глава была скорее обсуждением вопросов безопасности, чем разбором реализации кода. Причина в том, что потенциальные риски могут быть менее очевидными, чем сам код. Проблемы безопасности всплывают в местах, которые обычно не учитывают, например в логировании, мониторинге или требованиях к функционалу. Подробно о логировании мы поговорим в следующей главе, так как оно также помогает в отладке. Цель этой главы — показать, что безопасность нужно учитывать на каждом этапе разработки: от обсуждений с другими командами до написания кода. Вопросов при разработке и так много, но один из ключевых должен звучать так: «Безопасно ли это?» Если вы обнаружили уязвимость, то ИБ-специалист найдет еще больше. Ваши знания в этой области могут быть разными, но для эффективного использования инструментов вам не требуется погружаться в математические основы хеширования или шифрования.

ГЛАВА 9

Отладка бэкенда

Независимо от того, чем вы занимаетесь, вам потребуются надежные навыки и процессы отладки. Умение быстро анализировать возможные проблемы позволит эффективно выявлять первопричину. Не существует строго правильного или неправильного способа отладки, хотя есть определенные моменты, которые обязательно нужно проверять.

Отладка (debugging) — это искусство проявлять настойчивость, требующее дисциплины и системного подхода, поскольку процесс может быть извилистым и заводить в тупик. Зачастую код, вызывающий ошибку, не отличается сложностью. Это может быть изменение всего в одну строку, но поиск этой строки иногда занимает дни. Далее мы рассмотрим инструменты и стратегии, которые помогут максимально быстро добраться до корня проблем.

В этой главе я расскажу о:

- подробных сообщениях в логах;
- конфигурациях окружений;
- доступных инструментах;
- стратегиях трассировки ошибок (bug tracing);
- вариантах помощи коллегам в поиске ошибок.

Отладка — это та область, где менее опытные разработчики, столкнувшиеся с проблемами в своем коде, будут обращаться к вам за помощью. Это отличная возможность как для наставничества, так и для того, чтобы научиться чему-то новому. Некоторые ошибки оказываются слишком сложными для одного человека из-за масштабов системы. Со временем каждый член команды станет экспертом в своей части приложения, а значит, совместная работа позволит достигать результатов быстрее.

Подробные сообщения в логах

Один из способов помочь команде — писать информативные сообщения в логах. Количество логов в методе неограниченно, поэтому вам нужно решить, какая информация наиболее важна для всех. Давайте рассмотрим существующие логи в проекте и подумаем, как их улучшить.

Файл `stripe.service.ts` — отличное место для расширения логирования. В этот раз разберем метод `createProduct`, так как он содержит множество взаимодействующих частей. Здесь выполняются обновление базы данных и вызовы различных эндпоинтов стороннего сервиса. Если в этой цепочке вызовов и обновлений произойдет ошибка, как вы сможете отследить, что случилось? Давайте добавим логи и улучшим имеющиеся:

```
public async createProduct(product: CreateStripeProductDto) {
  this.logger.debug(
    `Начато создание продукта в Stripe с данными:
          ${JSON.stringify(product, null, 2)}`,
  );

  try {
    const productResponse = await this.stripe.products.create({
      name: product.name,
      description: product.description,
    });
    this.logger.log(
      `Ответ от stripe.products.create SDK:
            ${JSON.stringify(productResponse)}`,
    );

    this.logger.log(
      `Добавление информации о цене для product id: ${
        productResponse.id
      }, unit_amount: ${1009}, currency: ${'usd'} через stripe.prices.create SDK`,
    );
    const priceResponse = await this.stripe.prices.create({
      product: productResponse.id,
      unit_amount: 1009,
      currency: 'usd',
    });
    this.logger.log(`Ответ от stripe.prices.create SDK:
          ${JSON.stringify(priceResponse)}`);

    const productRecord = {
      stripeProductId: productResponse.id,
      name: productResponse.name,
      price: priceResponse.unit_amount,
    };

    this.logger.log(
      `Создание записи в таблице product с данными:
            ${JSON.stringify(productRecord)}`,
    );

    try {
      const [dbProduct] = await this.prisma.$transaction([
        this.prisma.product.create({ data: productRecord }),
      ]);
      this.logger.debug(`Создан продукт с id: ${dbProduct.id} в таблице product`);
```

```
    } catch (err) {
      this.logger.debug(
        `Откат транзакции для записи продукта: ${JSON.stringify(
          productRecord,
        )} с ошибкой: ${JSON.stringify(err)}`,
      );
    }
  } catch (err) {
    this.logger.error(
      `Ошибка Stripe с кодом статуса: ${err.status} и сообщением:
            ${JSON.stringify(err)}`,
    );

    throw new Error(`Ошибка Stripe с кодом статуса: ${err.status}`);
  }
}
```

Здесь представлено девять лог-сообщений, но при необходимости можно добавить больше. Обратите внимание, что каждый объект в сообщении преобразуется в строку. Без этого преобразования в логах будут появляться многочисленные записи `[Object object]`, что бесполезно для отладки. Вы также можете явно указывать окружение в логах, хотя многие инструменты, такие как Datadog (https://www.datadoghq.com) или Sentry (https://sentry.io/welcome), уже включают подобную функциональность. Однако если вам нужно быстро оценить ситуацию, дополнительная информация не помешает. Больше о мониторинговых инструментах вы узнаете в главе 12.

Здесь также используются различные уровни логирования. Основные уровни: *trace*, *debug*, *info*, *warn*, *error* и *fatal*. Уровень *debug* применяется для диагностики проблем или проверки корректности работы в тестовой среде. Уровень *trace* — это более детализированная версия *debug*: такие логи фиксируют все события в приложении и сторонних зависимостях. Они редко используются и нужны только для глубокого анализа конкретных участков кода.

Уровень *info* часто используется в логах для фиксации произошедших событий. С его помощью можно отслеживать авторизационные запросы, анализировать параметры вызовов API и контролировать состояние приложения. Однако в логах уровня *info* не должно быть подробностей, которые нужны для решения проблем.

Уровень *warn* применяется для регистрации неожиданных событий, не приводящих к остановке приложения (например, некорректный парсинг данных).

Уровень *error* следует использовать при сбоях, нарушающих ожидаемую работу приложения, включая ошибки 4xx и 5xx. Примеры:

- попытка оплаты через неработающий Stripe с переключением на резервный сервис;
- отказ одного из методов авторизации.

Как определить достаточный объем логирования

Дополнительные рекомендации по логам от Джеффа Грэма.

> Определение «правильного» количества логов может быть сложной задачей. С одной стороны, для отладки требуется максимум информации, но с другой — анализ длинного списка записей может быть утомительным. Полезно тестировать логи локально. Например, сможете ли вы диагностировать проблему с API, только читая логи? Также существуют инструменты для управления и поиска логов: CloudWatch, Splunk, Datadog, New Relic и многие другие.
>
> В логах могут неожиданно оказаться персональные данные или другая конфиденциальная информация: номера кредитных карт, API-ключи, адреса и т. д. Важно убедиться, что такие данные удалены или замаскированы в логах продакшена.
>
> Наконец, не превращайтесь в разработчика, который читает свои логи, только когда в продакшене происходит инцидент. Так вы рискуете обнаружить, что для отладки не хватает критически важных деталей.

Уровень *fatal* зарезервирован для критических сбоев бизнес-логики. Примеры:

- недоступность базы данных при запросе информации о заказе;
- полная неработоспособность всех методов входа в систему.

Правильное использование уровней логирования значительно упрощает работу. Возможность быстро различить логи уровня *error* и *debug* позволяет находить ответы быстрее, чем если все записывается на одном уровне. Проверяйте такие параметры, как версия API в запросе, проблемы с параметрами, источник трафика, а также время и дату генерации логов. Эти данные служат отправными точками для анализа ситуации. Таким образом вы сможете выявлять закономерности, например определенные ошибки, возникающие при передаче конкретных значений. Так проще обнаружить проблемы, связанные с одновременными запросами (concurrency issues), потому что вы сможете увидеть весь ход выполнения: когда запрос начался, когда завершился и что происходило между этими моментами.

Еще один аспект, требующий внимания, — вложенные блоки `try-catch`. Тщательно продумывайте порядок обработки ошибок, так как наличие таких блоков может привести к неожиданному подавлению ошибок.

Важно помнить, что при логировании ошибок из внешних источников необходимо включать исходную ошибку в лог. Кастомная обработка может легко скрыть настоящее сообщение об ошибке. При работе со сторонними сервисами полезно логировать как отправляемые запросы, так и получаемые ответы. Вот пример из предыдущего фрагмента, иллюстрирующий этот подход:

```
try {
    const productResponse = await this.stripe.products.create({
      name: product.name,
      description: product.description,
    });
    this.logger.log(
      `Ответ от stripe.products.create SDK:
            ${JSON.stringify(productResponse)}`,
    );
} catch (err) {
    this.logger.error(
      `Ошибка Stripe с кодом статуса: ${err.status} и сообщением:
            ${JSON.stringify(err)}`,
    );

    throw new Error(`Ошибка Stripe с кодом статуса: ${err.status}`);
}
```

Поскольку здесь реализована кастомная обработка ошибок с логированием, следует явным образом сохранять оригинальный ответ об ошибке, который приходит напрямую от Stripe и вызывает срабатывание блока `catch`. Именно это и делает лог ошибок в приведенном примере.

Здесь особенно полезным может оказаться тестирование — в идеале для каждого возможного пути выполнения кода должен быть написан модульный тест. Это также помогает выявлять баги на этапе разработки. Для каждого сценария ошибки должен быть тестовый кейс, что позволяет обнаружить, не подавляются ли сообщения об ошибках при кастомной обработке.

Анализируя эти логи, вы получите массу полезной информации. Фильтрация логов по времени возникновения проблем дает понимание того, как выполнялся код. Например, при ошибках создания продуктов в Stripe можно увидеть, когда выполнялись вызовы и какие данные спровоцировали ошибки. Могут обнаружиться ситуации с параллельными вызовами, приводящими к состоянию гонки (race condition) и, как следствие, к нарушению целостности данных.

Если у вас есть баг, а в коде все выглядит корректно, проверьте логи. Вот пример того, как могут выглядеть записи в логах:

```
[Nest] 90322  - 06/19/2024, 1:43:22 PM   DEBUG [OrdersService] Create order called
with:
{
  "user": {
    "id": 5,
    "roles": ["Customer"]
  },
  "order": {
    "total": 23.54,
    "userId": 5,
    "products": [
      {"id": "aperiam"},
```

```
      {"id": "ipsa"}
    ]
  }
}

[Nest] 90322  - 06/19/2024, 1:43:22 PM   ERROR [ExceptionsHandler] Cannot read
properties of undefined (reading 'includes')
TypeError: Cannot read properties of undefined (reading 'includes')
    at OrdersV1Controller.create (/Users/milecia/Repos/dashboard-server/src/orders/
orders.controller.ts:29:21)
    at /Users/milecia/Repos/dashboard-server/node_modules/@nestjs/core/router/
router-execution-context.js:38:29
    at processTicksAndRejections (node:internal/process/task_queues:95:5)
    at /Users/milecia/Repos/dashboard-server/node_modules/@nestjs/core/router/
router-execution-context.js:46:28
    at /Users/milecia/Repos/dashboard-server/node_modules/@nestjs/core/router/
router-proxy.js:9:17
```

На рис. 9.1 показан пример логов из Google Cloud Platform (GCP) — как одного из возможных инструментов, — но сейчас не будем на нем останавливаться. Более подробно инструменты логирования мы рассмотрим далее в книге.

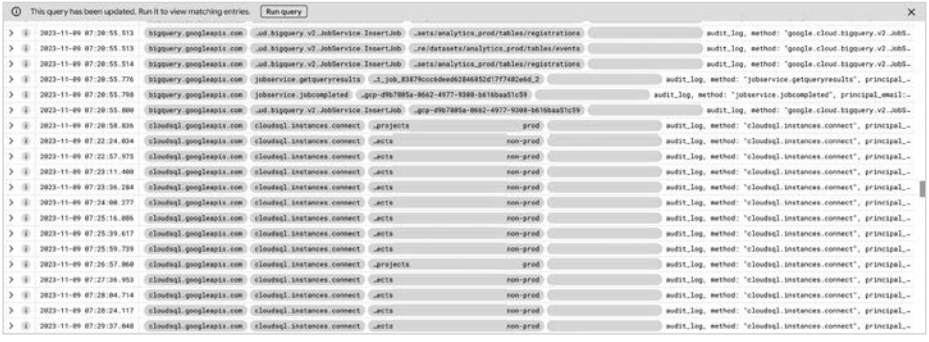

Рис. 9.1. Логи в Google Cloud Platform

Вот как могут выглядеть детали одного из таких логов:

```
{
  "protoPayload": {
    "@type": "type.googleapis.com/google.cloud.audit.AuditLog",
    "status": {},
    "authenticationInfo": {
      "principalEmail": "customersync@customerstore.iam.gserviceaccount.com",
      "serviceAccountDelegationInfo": [
        {
          "firstPartyPrincipal": {
            "principalEmail": "service-309022651100@serverless-robot-prod.iam.
gserviceaccount.com"
          }
```

```
        }
      ],
      "principalSubject": "serviceAccount:customersync@customerstore.iam.
gserviceaccount.com"
    },
    "requestMetadata": {
      "callerIp": "2600:1900:2000:a5::1:400",
      "requestAttributes": {
        "time": "2023-11-09T15:09:16.934724Z",
        "auth": {}
      },
      "destinationAttributes": {}
    },
    "serviceName": "cloudsql.googleapis.com",
    "methodName": "cloudsql.instances.connect",
    "authorizationInfo": [
      {
        "resource": "projects/customerstore/instances/customer-non-prod",
        "permission": "cloudsql.instances.connect",
        "granted": true,
        "resourceAttributes": {
          "service": "sqladmin.googleapis.com",
          "name": "projects/customerstore/instances/customer-non-prod",
          "type": "sqladmin.googleapis.com/Instance"
        }
      }
    ],
    "resourceName": "projects/customerstore/instances/customer-non-prod",
    "request": {
      "instance": "us-central1~customer-non-prod",
      "project": "customerstore",
      "@type": "type.googleapis.com/google.cloud.sql.v1.GenerateEphemeralCertRequest"
    },
    "response": {
      "ephemeralCert": {
        "kind": "sql#sslCert"
      },
      "@type": "type.googleapis.com/google.cloud.sql.v1.GenerateEphemeralCertResponse"
    }
  },
  "insertId": "-sae16ve1degi",
  "resource": {
    "type": "cloudsql_database",
    "labels": {
      "project_id": "customerstore",
      "database_id": "customerstore:customer-non-prod",
      "region": "us-central1"
    }
  },
  "timestamp": "2023-11-09T15:09:16.909393Z",
  "severity": "NOTICE",
  "logName": "projects/customerstore/logs/cloudaudit.googleapis.com%2Factivity",
  "receiveTimestamp": "2023-11-09T15:09:17.392870312Z"
}
```

При анализе следует обращать внимание на время возникновения записей в логах, сервисы, из которых они поступили, и содержащуюся в них информацию. При отладке сосредоточьтесь на логах, связанных с эндпоинтом или сервисом, где возникла проблема. Проверяйте возвращаемые статус-коды — они могут указать точное место ошибки в коде. Ищите сообщения об аутентификации, так как проблема может быть связана с ней. Далее проверьте, не передаются ли отсутствующие или невалидные параметры. Также анализируйте частоту появления ошибки в логах и попробуйте воспроизвести действие, которое генерирует соответствующий лог. Это поможет системно подойти к решению проблемы.

Файлы конфигурации окружений

Более подробно такие вещи, как CI/CD-пайплайны, инструменты и стратегии развертывания, а также другие связанные темы инфраструктуры мы будем рассматривать, начиная с главы 25. Сейчас же сосредоточимся на конкретных файлах конфигурации окружений, которые могут быть в вашей кодовой базе или в другом месте, к которому у вас есть легкий доступ, например в CircleCI (https://circleci.com). Следующий фрагмент кода — это пример файла конфигурации окружения, а на рис. 9.2 показаны эти переменные окружения в CircleCI:

```
# .env file
DATABASE_URL="postgresql://your_username:your_password@localhost:5432/dashboard"
STRIPE_SECRET_KEY="your_secret_stripe_key"
```

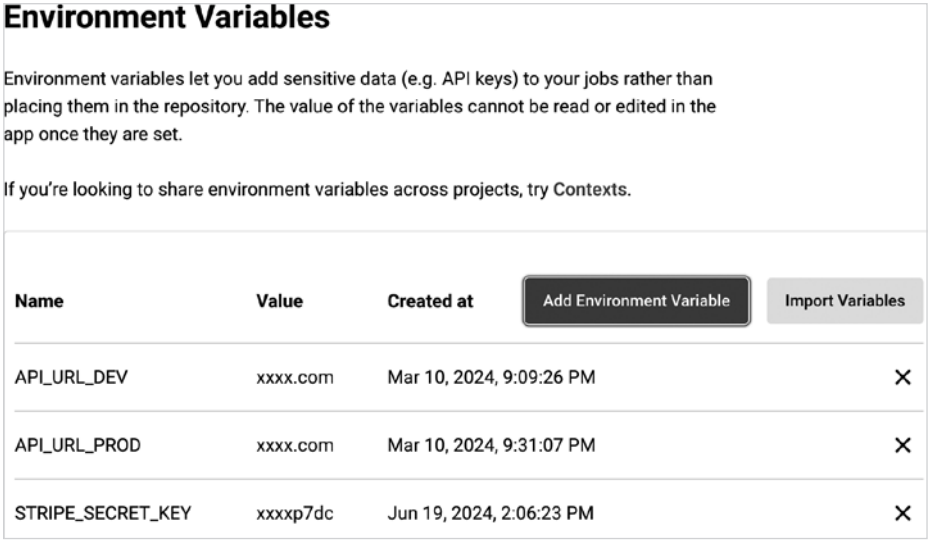

Рис. 9.2. Переменные окружения в CircleCI

При добавлении новых сторонних сервисов или потребности в новых переменных окружения убедитесь, что эти переменные действительно добавлены. Этот шаг легко упустить в процессе настройки всего остального. Подобные ошибки обычно быстро проявляются в CI/CD-пайплайне, и вы сможете добавить значения в нужные места. Это хороший момент для создания тикета, чтобы вся команда помнила о необходимости выполнения, а не только вы.

Убедитесь, что в запросе отсутствуют какие-либо роли на основе токена доступа (access token). Эта проблема может быть неочевидной — все зависит от того, как настроены ваши логи и обработка ошибок (error handling). Уточните, не были ли добавлены новые разрешения в рамках обновлений безопасности или нового функционала. Также убедитесь, что ваши сервисы получают корректные роли и разрешения в своих запросах.

Убедитесь, что ваши изменения развертываются в правильном окружении. Это особенно важно при первых нескольких развертываниях в новых окружениях. Проверьте, имеются ли необходимые *артефакты развертывания* и соответствуют ли хеши файлов (file hashes) ожидаемым значениям. Артефакт развертывания — это собранная версия вашего кода со всеми бандлами и метаданными, которая генерируется при компиляции кода для запуска на сервере. Их можно найти в вашем облачном сервисе, например в Amazon S3 bucket. *Хеш файла* — это уникальный строковый идентификатор каждого артефакта. Иногда может казаться, что обновленный артефакт развернут, но при проверке хеша оказывается, что это артефакт прежней версии.

Если вы не видите внесенных изменений, отладка усложняется, поскольку проблема не связана с изменением в вашем коде. Такое случается, и для детального анализа и проверки может потребоваться больше времени.

Если у вас сложный пайплайн развертывания, в рамках которого приложение перемещается между различными частями инфраструктуры, проверьте каждый этап. Вам могут попадаться сообщения об успешном выполнении операции, но при этом в сообщениях могут быть ошибки, возникающие в сервисах инфраструктуры. Просмотрите логи на каждом этапе, чтобы убедиться, что там нет никаких неприятных неожиданностей.

По мере углубления в отладку вы неизбежно столкнетесь с необходимостью анализа части инфраструктуры. Именно здесь могут размываться границы ответственности между командой разработчиков и командой DevOps. И той и другой следует изучить логи инфраструктуры, чтобы проверить, выполнилась ли команда или иное действие так, как ожидалось, и определить, заключается проблема в коде или в инфраструктуре. Так вы сможете решить, кто займется устранением ошибки и внесением изменений.

Это подводит нас к еще одной причине создавать и поддерживать актуальную схему архитектуры системы: вы точно будете знать, как все взаимосвязано, и сможете увидеть, где что-то могло пойти не так. Визуализация архитектуры напомнит о таких

вещах, как необходимость обновления токена доступа для фоновых и cron-задач. Кроме того, схема привлечет ваше внимание к другим частям системы, которые могут быть причиной проблемы, хотя изначально, вероятно, вы о них и не думали.

Стратегии отслеживания ошибок

Когда вы долго копаетесь в логах и коде, легко застрять на одном месте. Возможно, вы часами или даже днями смотрите на одни и те же файлы. Сделайте перерыв — прогуляйтесь или займитесь чем-то другим. Меня всегда удивляло, как быстро удавалось найти ошибку после того, как немного отвлечешься.

Если выйти из тупика не получается, переключитесь на другую часть приложения, которая может быть причиной проблемы. Обсудите задачу с коллегой — иногда пользу приносит даже короткий разговор. Мне лично помогает такое правило: первые 30 минут пробую отладить самостоятельно, а если прогресса нет, иду за помощью. Отвлечение внимания и свежий взгляд со стороны часто заставляют по-другому увидеть процесс отладки. Можно также использовать форумы вроде Stack Overflow или Discord-серверы, только сначала обезличьте код, чтобы не раскрыть корпоративную информацию. Эти методы сделают отладку максимально эффективной.

Отладка — это именно та область, где могут пригодиться ИИ-инструменты типа ChatGPT. Иногда достаточно ввести вопрос в чат, и ответ может оказаться таким же полезным, как обсуждение вопроса с другим разработчиком. Если правила вашей организации по защите кода позволяют, вы даже можете вставить фрагмент кода в сервис и получить ценные рекомендации по идеям дальнейших действий.

Используйте `console.log`! Это самый простой инструмент отладки, который у вас есть. Вы можете логировать каждую строку кода, если это помогает понять происходящее. Сообщения в консоли показывают неожиданные вызовы или преобразования данных в реальном времени. Кроме того, смело используйте коды, написанные «на коленке», чтобы воспроизвести проблему, проверить гипотезу или применить потенциальное исправление. При поиске первопричины код не обязательно должен соответствовать стандартам для пул-реквеста. Сначала проверяйте идеи, потом приводите код в порядок.

Вот пример кода «на коленке» для проверки работоспособности эндпоинта. Можно постепенно раскомментировать участки и вручную изменять входные данные пользователя (`user`) и заказа (`order`), чтобы выявить источник ошибки.

```
// if (!user) {
    //   throw new HttpException('Unauthorized', HttpStatus.UNAUTHORIZED);
    // }
    // if (!user.roles.includes('Customer')) {
    //   throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
    // }
```

```
// if (!order) {
//   throw new HttpException('No order data', HttpStatus.BAD_REQUEST);
// }
// if (order.products.length === 0) {
//   throw new HttpException('No products in order', HttpStatus.CONFLICT);
// }
// if (!order.total) {
//   throw new HttpException('No order total', HttpStatus.CONFLICT);
// }

try {
  // const newOrder = await this.ordersService.createOrder(order);
  const newOrder = {
    id: 35354252,
    total: 1342.24,
    createdAt: new Date(),
    updatedAt: new Date(),
    products: [],
    userId: 7,
  }
  return newOrder;
} catch (err) {
  throw new HttpException('Something happened', HttpStatus.NOT_FOUND);
}
```

Знание инструментов для проверки гипотез придаст вам уверенности при исследовании различных аспектов. Если вы подозреваете проблему в запросе или ответе, попробуйте использовать Postman (https://postman.com) для отправки запроса, как это сделал бы клиент. Когда нужно проверить корректность обновления данных, воспользуйтесь таким инструментом, как pgAdmin для Postgres (https://oreil.ly/LsJrb). Он предоставляет быстрый графический интерфейс для прямого взаимодействия с базой данных, что полезно во время парного программирования.

Создайте тестовые данные прямо в коде и запустите его локально. Иногда достаточно быстро набросать код для воспроизведения бага — в процессе вы можете случайно решить проблему. Такой подход помогает выявлять интересные сторонние баги, когда вы видите реальный ответ. Например, вы можете столкнуться с неожиданным преобразованием данных из-за некорректного вызова функции.

Отладка бэкенда иногда затрагивает вопросы работы инфраструктуры. Встречаются ситуации, когда данные не удалились из-за тихо провалившегося события два дня назад. Это сложный процесс, влияющий на множество частей системы, и его отладка требует времени. Освойте эту практику: изучайте сервисы во время разработки функционала и исправления багов. Тогда при возникновении серьезной проблемы вы будете уверенно ориентироваться в инфраструктуре.

Перепроверяйте бизнес-логику. Иногда система работает именно так, как должна, но возникают конфликты на уровне бизнес-правил. Обсуждая проблемы с продуктовой командой, убедитесь, что бизнес-правила корректно реализованы в коде. Тут

могут быть разные тонкие моменты, поскольку требуется интуитивно понимать работу продукта, и здесь важно четко различать, где баг, а где фича.

Подробнее об обработке инцидентов мы поговорим в главах 12 и 23, но стоит кратко упомянуть это здесь. Держите коллег в курсе проверенных гипотез, особенно при критических багах в продакшене, влияющих на пользователей. Делитесь выводами по мере исключения возможных причин. Это поможет систематизировать выполненные действия и разобраться, кто может помочь.

Помощь другим разработчикам в отладке

Умение помогать другим в отладке их проблем — ценный профессиональный навык. Здесь можно применить несколько подходов. Первый — научный метод: вы формируете гипотезу о природе проблемы, создаете тест и затем запускаете его. Хороший пример — ошибка доступа (forbidden error), например:

```
if (!user.roles.includes('Customer')) {
  throw new HttpException('Доступ запрещен', HttpStatus.FORBIDDEN);
}
```

Вы и другой разработчик можете выдвинуть гипотезу, что у пользователя отсутствует роль `Customer`. Затем стоит проверить эту гипотезу, вызвав эндпоинт локально через Postman и передав данные пользователя с подходящей ролью. После выполнения теста просмотрите терминал, где запущено приложение, на наличие ошибок или сообщений. Этот процесс можно повторить с другой гипотезой. Как вариант: сформулируйте несколько гипотез, проверьте их по отдельности и поделитесь результатами.

Другой метод — построение детальной блок-схемы, которая отображает точную последовательность событий и участков кода, затронутых ошибкой. Вы и разработчик можете использовать инструменты вроде Miro или даже просто Google Slides для быстрой визуализации происходящего. На рис. 9.3 представлен пример такой блок-схемы для той же ошибки, что и в предыдущем примере.

Следующий подход — вычисление проблемного компонента. Попытайтесь локализовать ошибку в контроллере или сервисе. Если она в контроллере, декомпозицию можно прекратить. Если в сервисе — определите, связана ли проблема с кодом сервиса, базой данных или сторонним API. Продолжайте углубляться по цепочке до тех пор, пока не найдете модуль приложения, являющийся наиболее вероятным источником ошибки.

Еще один метод совместной отладки — *проговаривать мысли вслух* (rubber-ducking[1]) (https://oreil.ly/1i554). В этом методе надо проговаривать каждую строку

[1] Метод утенка (Rubber Duck Debugging) — это техника отладки кода, при которой программист объясняет свою проблему вслух воображаемой собеседнице — резиновой уточке. — *Примеч. ред.*

кода неодушевленному предмету или коллеге. Просто дайте разработчику возможность изложить проблему, пока вы его слушаете. Удивительно, как много ошибок решается просто в процессе такого обсуждения. Формулировка проблемы обычными словами часто приводит к инсайтам или делает очевидным то, что упускалось при чтении «про себя». Метод работает как в голосовом формате, так и в переписке. Главное — побудить человека детально описать ошибку и свои действия. Порой вам даже не потребуется что-либо отвечать — решение найдется за несколько минут!

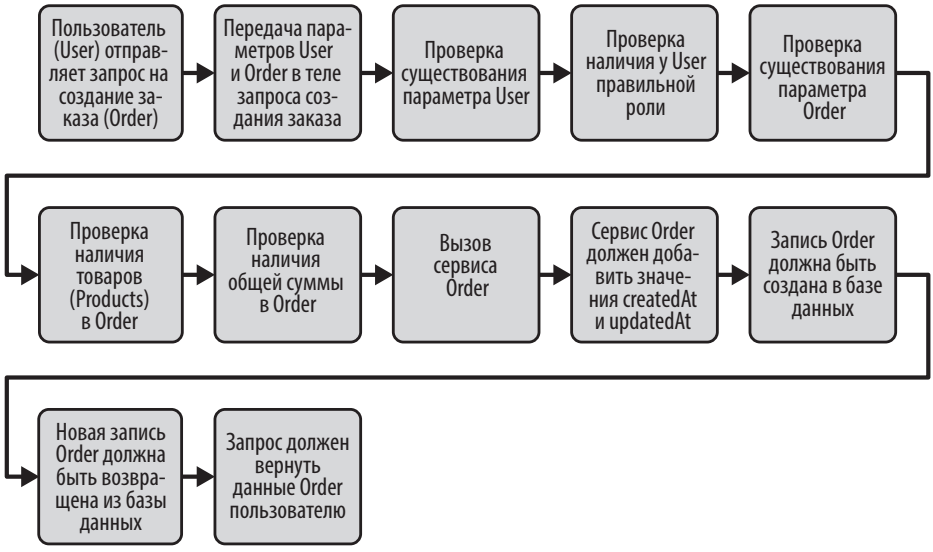

Рис. 9.3. Диаграмма последовательности отладки

Последний подход, который я рассмотрю, — это помощь в использовании инструментов, таких как ChatGPT (https://chat.openai.com) или Copilot (https://oreil.ly/y0kZy). Эти инструменты могут выступать в роли автоматизированного собеседника и наставника, поскольку они анализируют код на предмет несоответствий. Они также заставляют задуматься о том, как описать проблему, чтобы *получить* помощь. Подобные инструменты ИИ помогают упростить проблему и дают краткое описание того, что происходит с кодом.

Чек-лист по отладке

Это не исчерпывающий список всех возможных проверок при отладке бэкенда, но он довольно обширный. Основная цель — выделить конкретные моменты, на которые стоит обратить внимание на каждом этапе. Это похоже на точное разрезание торта. Надеюсь, этот чек-лист станет хорошей отправной точкой, если вы застрянете в процессе отладки.

- Проверка логов приложения:
  - Есть ли ошибки? Если да, то откуда они поступают?
  - Можно ли определить время, когда проблема возникла?
  - Удалось ли найти полезные трассировки стека вызовов?
  - Если в ошибках указана строка кода, перешли ли вы к ней?
- Проверка токена доступа (access token):
  - Не истек ли срок его действия?
  - Имеет ли запрашивающая сторона корректные разрешения?
  - Не поврежден ли токен (например, кто-то пытался его изменить)?
- Проверка запроса:
  - Присутствуют ли все необходимые параметры?
  - Соответствуют ли типы параметров ожидаемым?
  - Возникает ли проблема CORS (Cross-Origin Resource Sharing) (https://oreil.ly/cVGm5)?
  - Проверены ли заголовки?
- Проверка ответа:
  - Был ли отправлен ожидаемый тип ошибки?
  - Данные пришли в правильном формате?
  - Происходят ли преобразования данных перед отправкой ответа?
  - Вызываются ли сторонние сервисы перед формированием ответа?
- Проверка базы данных:
  - Обновлялись или запрашивались ли ожидаемые поля?
  - Когда в последний раз обновлялись данные?
  - Каков результат выполнения сырого SQL-запроса по сравнению с запросом из приложения?
- Проверка окружения:
  - Добавлены ли новые переменные окружения?
  - Обновлены ли существующие переменные окружения?
  - Внесены ли переменные окружения во все инструменты инфраструктуры и использующий их код?
  - Производил ли сторонний сервис ротацию значений?
  - Выполняется ли развертывание в правильное окружение?
  - Можно ли проверить версию артефакта?

    - Обращались ли за помощью к коллеге из команды DevOps для совместной работы?
- Проверка сторонних сервисов:
    - Вышли ли новые версии?
    - Изменились ли названия или типы полей?
    - Можно ли сверить типы запросов и ответов в документации?
    - Были ли изменения в панели управления?
    - Требуется ли новый API-ключ?
- Проверка кода:
    - Все ли версии пакетов актуальны?
    - Проверяли ли методы построчно?
    - Пробовали ли максимально упростить код (как в примере «на коленке») и постепенно добавлять функциональность обратно?
    - Есть ли блоки `try-catch`, которые могут скрывать ошибки?
    - Как обрабатываются ошибки базы данных?
    - Корректна ли бизнес-логика?
    - Есть ли другие части кода, взаимодействующие с этой?

По мере приобретения опыта вы можете составить собственный список методов отладки. У меня есть такой список, и он помогает быстрее исключать возможные причины и избавляет от необходимости тратить на это умственную энергию. Проверка всех указанных аспектов утомительна, но зато это исчерпывающий способ исключить корневые причины.

Заключение

Отладка — одна из областей разработки, которая требует настойчивости. Возможно, у вас будут иногда складываться ситуации, где один разработчик исправляет ошибку всего за полчаса, а вы — за несколько дней. Просить о помощи — нормально. Это показывает, что вы понимаете, как другие разработчики могут помочь устранить потенциальные источники ошибок.

По мере роста вашего приложения будет расти и сложность отладки. Не существует идеального способа поиска ошибок. Иногда вам придется по-быстрому набрасывать код, писать сообщения в консоли или даже менять данные прямо в БД. Помните, что это одна из творческих сторон вашей работы. Любой подход годится, если не ломает работу других частей системы!

ГЛАВА 10

Производительность бэкенда

Теперь, когда ваше приложение готово к продакшену, пора задуматься о возможных улучшениях производительности. На этом этапе вы начнете анализировать метрики, настраивать инфраструктуру и писать конфигурационные файлы. Изменения в коде также могут повысить производительность, поэтому никогда не недооценивайте качественный рефакторинг.

Подход к оптимизации производительности сильно зависит от вашего приложения. Чтобы определить, где и как вносить улучшения, потребуется глубокое понимание работы приложения, его текущего состояния и ожидаемого роста. Получив эти данные, вы сможете оценить компромиссы между доступными вариантами. Далее мы рассмотрим несколько аспектов, которые стоит изучить по мере роста пользовательской базы, в частности, мы поговорим о:

- метриках;
- оповещениях и мониторинге;
- кэшировании;
- продуктовой специфике;
- прочих способах повышения производительности.

Вообще работа над улучшением производительности — это волнующий этап: он означает, что продукт активно используется. На этом этапе большую пользу приносят совещания, поскольку по их итогам компания выделяет средства на реализацию принятых решений. Стейкхолдеры видят продукт под другим углом, и это помогает принимать решения о том, какой подход лучше в той или иной ситуации.

Метрики

Чтобы понять, нужно ли заниматься оптимизацией производительности, сначала выясните, почему эта тема вообще возникла. Пользователи жаловались на медленную загрузку? Сервер падал под наплывом пользователей? Оптимизация также может быть и превентивной мерой. Если вы выпускаете долгожданное

приложение, ваши серверы могут столкнуться с уровнем нагрузки, который вы ранее считали маловероятным.

Поэтому, если вас терзают сомнения или вам нужно обосновать дополнительные затраты на разработку, обратитесь к данным. Изучите логи, чтобы выявить общие тенденции. Возможно, вы обнаружите, что один эндпоинт получает значительно больше запросов, чем остальные. Или заметите, что определенная группа эндпоинтов активнее всего используется в конкретное время суток. Такой бизнес-анализ помогает определить, какие улучшения принесут наибольшую пользу.

Вот некоторые метрики бэкенда, которые вы можете встретить:

- средняя задержка приложения (среднее время, необходимое приложению для ответа на запрос, исключая время обработки);
- стандартное отклонение задержки для ответов;
- минимальная и максимальная задержка;
- P90 задержка (наиболее медленный ответ в 90-м процентиле самых быстрых запросов);
- количество запросов в секунду;
- соотношение ввода-вывода данных (метрика ввода данных — размер входящих полезных нагрузок запросов, метрика вывода данных — размер исходящих полезных нагрузок ответов);
- пиковое время ответа (PRT, peak response time; определение показателей самого длительного времени ответа среди всех запросов к серверу);
- использование аппаратных ресурсов, таких как оперативная память (RAM) и дисковое пространство;
- количество используемых потоков для всех запросов (определяет, сколько параллельных запросов происходит одновременно);
- нагрузка на сервер (измеряет среднее количество потоков, ожидающих процессорного времени в заданном временном диапазоне или выполняющихся в этом же диапазоне);
- доступность сервера (процент времени, в течение которого сервер доступен для использования);
- частота HTTP-ошибок сервера (как часто возникают определенные ошибки);
- Apdex (открытый стандарт для агрегирования удовлетворенности пользователей в виде взвешенного среднего балла).

Обычно бизнес выбирает определенный набор метрик под свои нужды. Вам надо будет уверенно работать с различными сервисами и находить подобную информацию в дашбордах вашей облачной платформы. Примеры дашбордов Splunk на logit.io (https://oreil.ly/sH3wF) — хорошие образцы того, что стоит искать при анализе продукта. В них выделены ключевые метрики, наиболее

значимые для пользователей, такие как критические ошибки (severity error) и производительность сети. Вам надо будет делать нечто подобное для своего продукта, когда вы будете определять, какие метрики важны. На рис. 10.1 и 10.2 приведены примеры метрик, которые можно встретить в дашборде GCP (Google Cloud Platform).

Дополнительные сведения о метриках производительности

Рассмотренные в этом разделе метрики содержат множество технических нюансов. Поэтому обратимся за разъяснениями к Итану Брауну.

Что касается P90 задержки:

> P90 задержка — это значение, которое разделяет запросы на 90% (с задержкой на уровне или выше определенного порога) и 10% (с более медленным откликом). Например, при P90 задержке в 100 мс 90% запросов выполняются быстрее 100 мс, а оставшиеся 10% — медленнее.

О показателе PRT:

> Термин «задержка» здесь явно относится к задержке на транспортном уровне (без учета времени обработки), тогда как PRT включает обработку. Максимальная задержка и PRT — это разные показатели.

О количестве потоков:

> По умолчанию Node — это однопоточный сервер. Если вы специально не настроите его на использование worker threads, он будет однопоточным. Конфигурация сервера может включать несколько процессов Node с балансировкой нагрузки, но стандартный подход — горизонтальное масштабирование (scaling out), а не вертикальное (scaling up). Хотя однопоточный процесс Node способен обрабатывать параллельные запросы через event loop (модель событийного цикла), это не отражается в статистике потоков ОС.

Поскольку эти метрики наиболее наглядны, когда приложение уже в продакшене, вам будет полезно увидеть реальные цифры в инструментах, используемых в вашей организации или в текущем проекте. Если у вас есть возможность, изучите в ближайшее время все серверные инструменты, которыми вы пользуетесь. Если не знаете, где найти те или иные данные или что означают определенные метрики, не стесняйтесь спросить у коллег. Освойтесь с теми областями, которые пока менее знакомы. Не обязательно становиться экспертом, но важно владеть достаточным объемом информации, чтобы работать эффективно.

Еще несколько возможных направлений оптимизации включают оптимизацию алгоритмов (algorithmic performance) и оптимизацию запросов к базе данных

(database query optimization). *Оптимизация алгоритмов* (https://oreil.ly/EEEIz) обычно наиболее актуальна для приложений с высокой интенсивностью вычислений, таких как системы обработки данных или приложения, работающие с расчетами в реальном времени (например, видеоигры). В этом случае следует учитывать метрики временной (time complexity) и пространственной сложности (space complexity). *Оптимизация запросов к базе данных* (https://oreil.ly/_XTON) подразумевает оптимизацию SQL-запросов для повышения производительности и эффективности работы с БД. Такая оптимизация может включать сокращение времени чтения/записи или оптимизацию выборки данных.

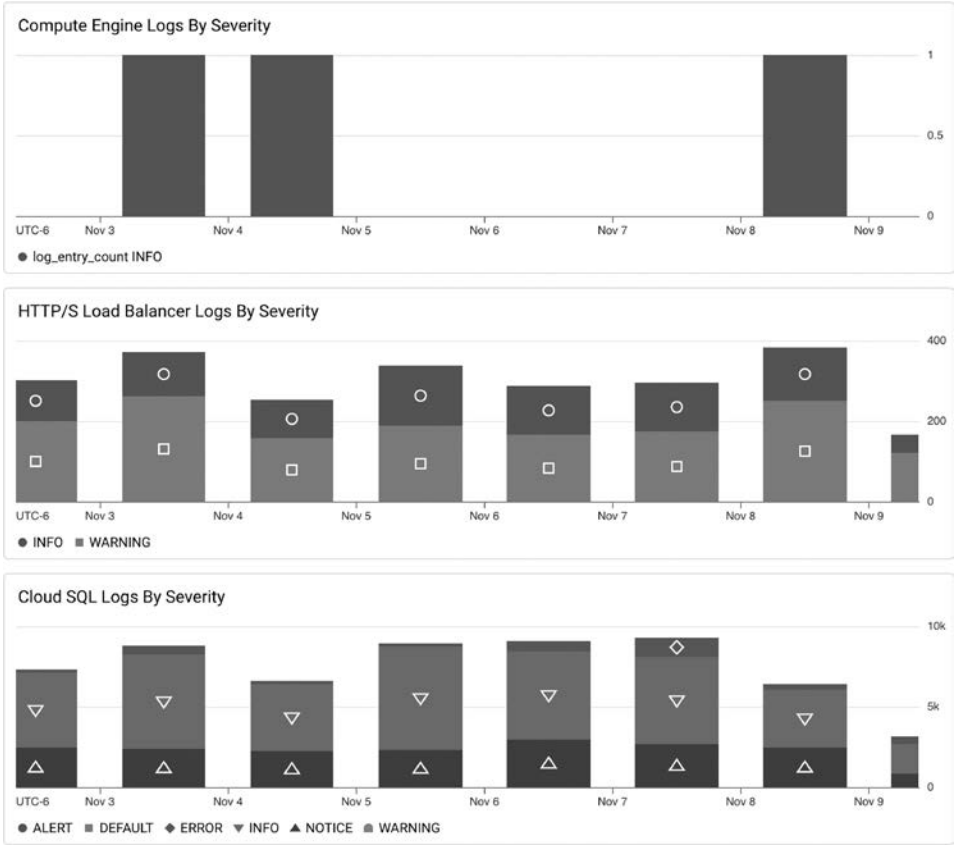

Рис. 10.1. Метрики GCP

Я не буду подробно останавливаться на этих двух аспектах, поскольку мы тогда немного опередим события. Однако по мере выявления закономерностей и тенденций важно своевременно информировать команду. Так вы сможете предотвратить проблемы с производительностью до их эскалации. Отслеживание метрик также предоставит дополнительные данные для оптимизации рабочих процессов.

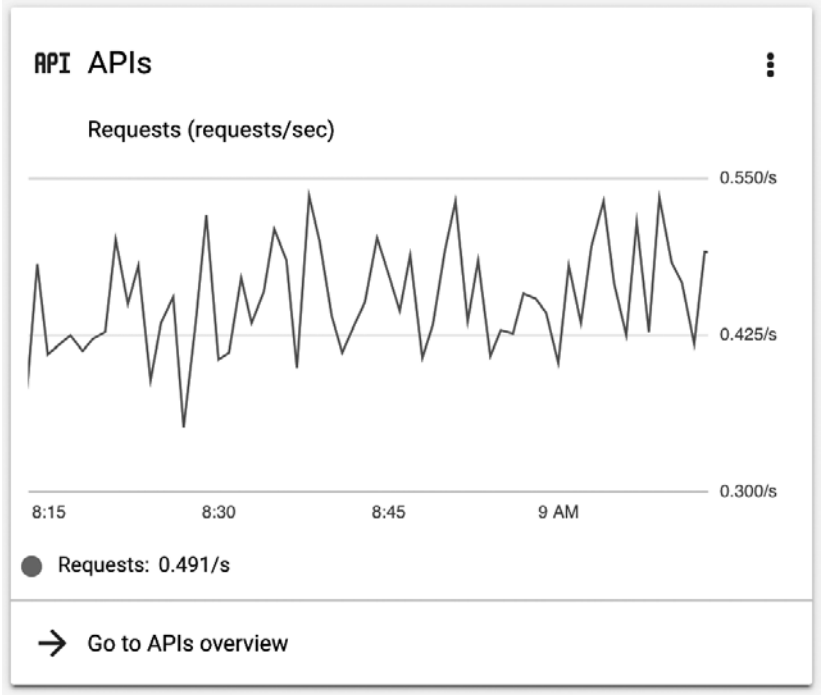

Рис. 10.2. Количество запросов к API в секунду в GCP

Оповещения и мониторинг

Поскольку вы уже собираете метрики, логично добавить автоматизацию в виде мониторинга и оповещений. Мониторинг поможет отслеживать достижение метриками заданных пороговых значений, а оповещения уведомят вас о выходе за эти значения. Установив пороги для метрик, вы сможете анализировать частоту их приближения к максимальным значениям — это укажет, какие части приложения требуют доработки.

Долгосрочный мониторинг метрик выявит сезонные и годовые тенденции. На основе этих данных можно принимать решения, значительно сокращающие расходы организации на облачные ресурсы. Вы сможете определить периоды повышенной нагрузки и масштабировать инфраструктуру соответственно.

Полезной практикой для вашей команды может стать проведение ежеквартальных или полугодовых встреч, посвященных обсуждению этих метрик. Это отличный способ познакомить junior-разработчиков с процессами, происходящими «за кулисами» кода, а также дать всем представление о направлениях для технических улучшений. Можно также пригласить на такие встречи продуктовую команду, чтобы они увидели, с какими техническими аспектами приложения пользователи взаимодействуют чаще всего.

Еще один вариант использования мониторинга — запуск событий при достижении метрикой определенного диапазона. Вы можете использовать встроенные сервисы мониторинга и оповещений вашего облачного провайдера, такие как Amazon CloudWatch (https://oreil.ly/goYNU), Azure Monitor (https://oreil.ly/8ig8-) и Google Cloud Monitoring (https://oreil.ly/Nt7vi). Также популярны сервисы вроде Zapier (https://zapier.com), n8n (https://n8n.io) и Make (https://www.make.com), которые интегрируются со множеством инструментов. Они позволяют автоматически запускать сложные сценарии при достижении метриками заданных пределов. Например, можно настроить автоматическое масштабирование ресурсов, отправку email- и Slack-уведомлений команде разработчиков, обновление сообщения на сайте и изменение значения в базе данных. Независимо от сложности вашего сценария автоматизации, важно четко понимать, что именно происходит.

Оповещения помогают несколькими способами. Главное преимущество — в продакшене вы узнаете о проблемах сразу при их возникновении благодаря сообщениям в специальных чатах или каналах с соответствующими членами команды. Это относится к практике ChatOps (https://oreil.ly/Tn7Lw), где инструменты автоматизируют коммуникацию между ресурсами и людьми (рис. 10.3). Оповещения распределяют ответственность и повышают осведомленность команды о проблемах. Важно отслеживать, как можно расширить возможности команды. Убедитесь, что все понимают принцип работы автоматизации и отслеживаемые метрики.

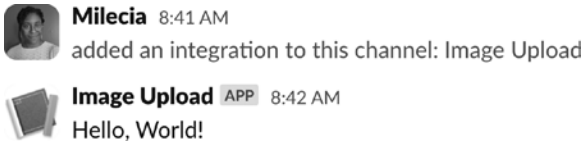

Рис. 10.3. Интеграция бота в Slack

После настройки мониторинга и оповещений, а также осознания, где находятся метрики, задокументируйте процесс и проведите краткий созвон с командой для разъяснений. Это быстрый способ повысить уровень осведомленности команды по мере изучения новых аспектов. Команда, в свою очередь, поможет вам развиваться через задаваемые вопросы. Вопросы всегда полезны, поскольку указывают на области знаний, в которые следует углубиться.

Работа с оповещениями — это тоже искусство. Важно отправлять только самые критичные уведомления, иначе все они рискуют стать «белым шумом». По мере изменения метрик потребуется актуализировать настройки оповещений. Выделите время на анализ текущих уведомлений: определите наиболее частые срабатывания и внесите соответствующие корректировки. Кроме того, стоит распределять оповещения по разным каналам в зависимости от их важности. Например, критические оповещения из продакшена можно направлять в Slack-канал и дублировать email-рассылкой для команды. Менее важные уведомления

могут поступать в другой Slack-канал без email-уведомлений. Подробнее об этом мы поговорим в главе 12.

После внедрения этих механизмов и нескольких недель тестирования можно приступать к экспериментам с оптимизацией производительности.

Кэширование

В любой организации обычно отслеживают несколько метрик, например задержку. Как правило, организации стремятся минимизировать это значение, чтобы обеспечить пользователям максимально короткое время отклика. Один из способов улучшения — серверное кэширование или кэширование базы данных. Кэширование базы данных подразумевает выделенное хранилище для сохранения ответов серверов, перед которыми оно расположено, что снижает задержку за счет уменьшения количества запросов к базе данных.

Нагрузка (load) — еще одна измеряемая метрика, но она сложнее задержки. При serverless-подходе метрика нагрузки может быть неактуальна либо могут быть важны другие показатели. Если вы управляете собственными серверами для обработки совокупной нагрузки, ваша цель — найти баланс. Слишком низкая постоянная нагрузка означает неэффективную оплату неиспользуемых ресурсов, а постоянная высокая нагрузка нежелательна. Поэтому необходимо резервировать ресурсы для пиков использования, которые сложно предсказать.

Сервер по возможности сначала обращается к кэшу при наличии такового. В кэше хранятся ответы для наиболее часто запрашиваемых эндпоинтов. Набор этих эндпоинтов обычно определяется в процессе мониторинга. Значения ответов попадают в кэш либо через заданные интервалы времени, либо при первом запросе в определенный период. Таким образом пользователи получают актуальные данные, а не устаревшую информацию.

Если данные запроса найдены в кэше, сервер вообще не обращается к базе данных, что экономит время на формирование ответа. Именно так кэширование повышает производительность сервера — оно гарантирует, что сервер базы данных вызывается только при необходимости, а не при каждом запросе (рис. 10.4). Такой тип кэширования также называют *кэшированием базы данных*, поскольку он сокращает количество выполняемых запросов.

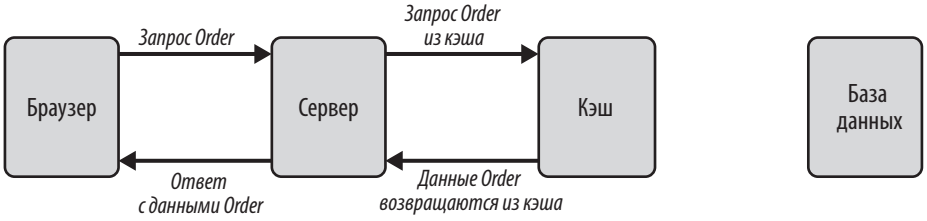

Рис. 10.4. Попадание в кэш (cache hit)

При использовании кэша крайне важно не хранить в нем персональные данные (PII). Это уязвимость безопасности, поскольку всегда есть вероятность, что злоумышленник атакует кэш вместо сервера, если выяснит, какая система используется. Отравление кэша (cache poisoning) (https://oreil.ly/7A7zh) — это уязвимость, позволяющая злоумышленникам внедрять вредоносный контент в кэш, к которому обращаются пользователи. Пользователи будут получать этот контент до тех пор, пока кэш не будет очищен.

Иногда запрашиваемых данных нет в кэше или они устаревают и удаляются. В таких случаях обычно происходит промах кэша (cache miss), как показано на рис. 10.5. Это означает, что запрос проходит через кэш и достигает базы данных. Ответ сервера сохраняется в кэше, а затем отправляется в браузер.

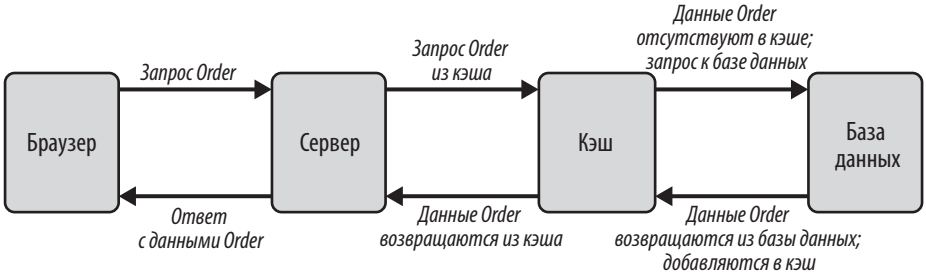

Рис. 10.5. Промах кэша

Стратегии кэширования

Существует несколько стратегий кэширования, которые помогают балансировать между актуальностью данных и скоростью ответа. К ним относятся read-through (чтение через кэш), write-through (сквозная запись), write-back (отложенная запись) и cache-aside (кэш рядом).

При стратегии *чтения через кэш* (рис. 10.6) сначала всегда проверяется наличие данных в кэше. Если данные отсутствуют, запрос перенаправляется в бэкенд для их получения из базы данных, после чего они сохраняются в кэше. Поскольку механизма обновления данных в кэше нет, эту стратегию лучше использовать в случаях, когда база данных обновляется редко или запросы выполняются очень медленно. Обратите внимание: стратегия чтения через кэш не влияет на обработку запросов на обновление или создание данных — они по-прежнему отправляются напрямую на сервер и в базу данных.

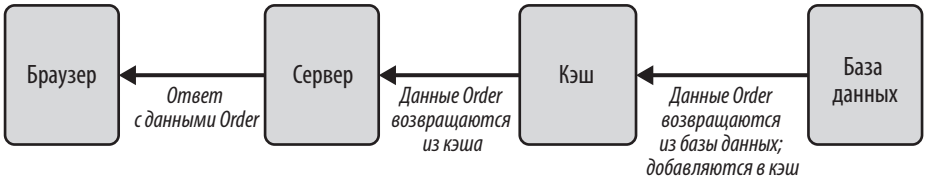

Рис. 10.6. Стратегия чтения через кэш

Стратегия *сквозной записи*, показанная на рис. 10.7, позволяет записывать данные одновременно в кэш и базу данных из поступающих запросов. Это гарантирует актуальность данных, но может замедлить первоначальную операцию записи, поскольку приходится ждать завершения записи в базу данных. Обычно эту стратегию реализуют вместе со стратегией сквозного чтения для достижения наилучшей производительности.

Рис. 10.7. Стратегия сквозной записи в кэш

Также можно использовать стратегию *отложенной записи*, как показано на рис. 10.8. Она похожа на сквозную запись, но с тем отличием, что выполняется несколько записей в кэш перед обновлением базы данных. Это ускоряет обработку пользовательского ввода и позволяет выполнять пакетные обновления базы, однако важно избегать серьезных рассогласований данных или их потери, если с кэшем что-то произойдет до синхронизации с базой. Отложенная запись полезна для сценариев с высокой интенсивностью записи, где мгновенная доступность данных в базе некритична, например при обработке изображений или аудио.

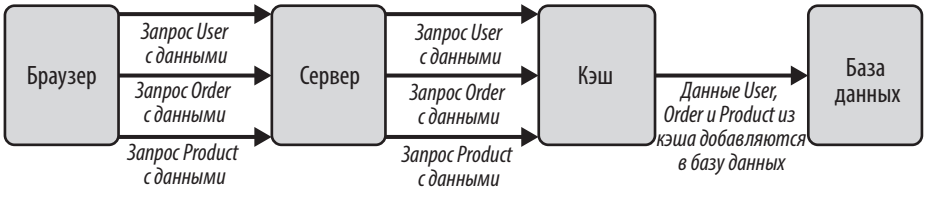

Рис. 10.8. Стратегия отложенной записи

Последняя стратегия — подход *«кэш-рядом»* (cache-aside), показанный на рис. 10.9. В этом случае приложение самостоятельно управляет кэшем. При запросе данных приложение сначала проверяет кэш. Если данных в кэше нет, оно получает их из базы данных и сохраняет в кэш. Этот подход достаточно универсален. Основная сложность здесь — поддержание актуальности данных в кэше. Эту стратегию стоит рассмотреть, когда спрос на ресурсы непредсказуем и можно загружать данные по требованию. Также ее можно использовать, когда кэш не поддерживает операции сквозного чтения и сквозной записи.

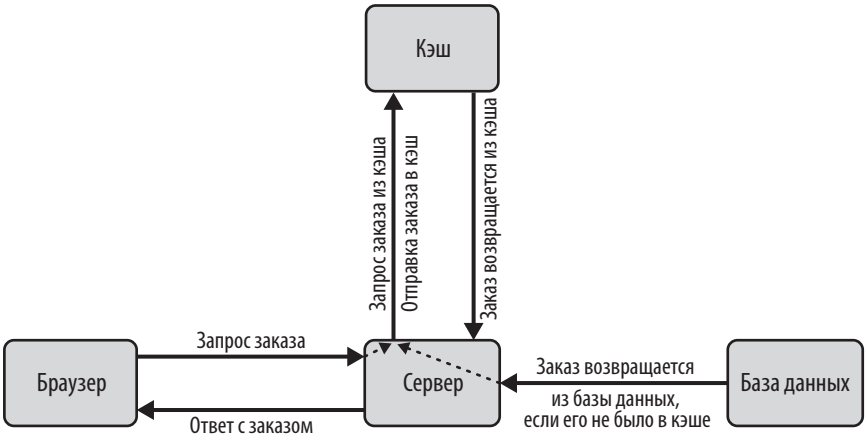

Рис. 10.9. Стратегия «кэш-рядом»

Типы кэширования

Существуют разные типы кэширования: как самостоятельно реализуемые, так и с использованием специализированных инструментов; обычно оба варианта работают в оперативной памяти (in-memory). Кэширование в оперативной памяти хранит данные в RAM сервера. Этот подход полезен при работе с большими объемами редко изменяющихся данных, например со списком товаров. Такой список может загружаться при первом посещении сайта пользователем, а затем кэшироваться в памяти. Реализация in-memory кэширования обычно выполняется быстро и обеспечивает минимальное время отклика. Главный недостаток — данные теряются при перезагрузке системы или очистке оперативной памяти.

Среди распространенных инструментов кэширования — Redis (https://redis.io) и Memcached (https://memcached.org). Используя такие инструменты, вы можете не заботиться о деталях реализации. Если вы разрабатываете приложение для пользователей, живущих по всему миру, тогда такой тип кэширования вполне оправдан. Вы можете разместить серверы кэша в разных регионах, чтобы обеспечить быстрое время отклика с помощью распределенной системы кэширования.

Распределенные серверы кэша с этими инструментами — экономически эффективный способ обеспечить доступность данных по всему миру. Однако есть и недостатки: могут возникнуть проблемы с согласованностью данных, а система станет сложнее в управлении. При несоответствиях данных разные пользователи могут видеть разные наборы информации. Кроме того, с ростом распределенной системы становится сложнее отслеживать проблемы и управлять конфигурациями.

Клиентское кэширование также существует, но мы вернемся к нему в главе 20. После того как вы определились с интеграцией кэширования в архитектуру, сделайте паузу и оцените решение под другим углом. На рис. 10.10 показан пример того, как можно добавить Redis в вашу архитектуру.

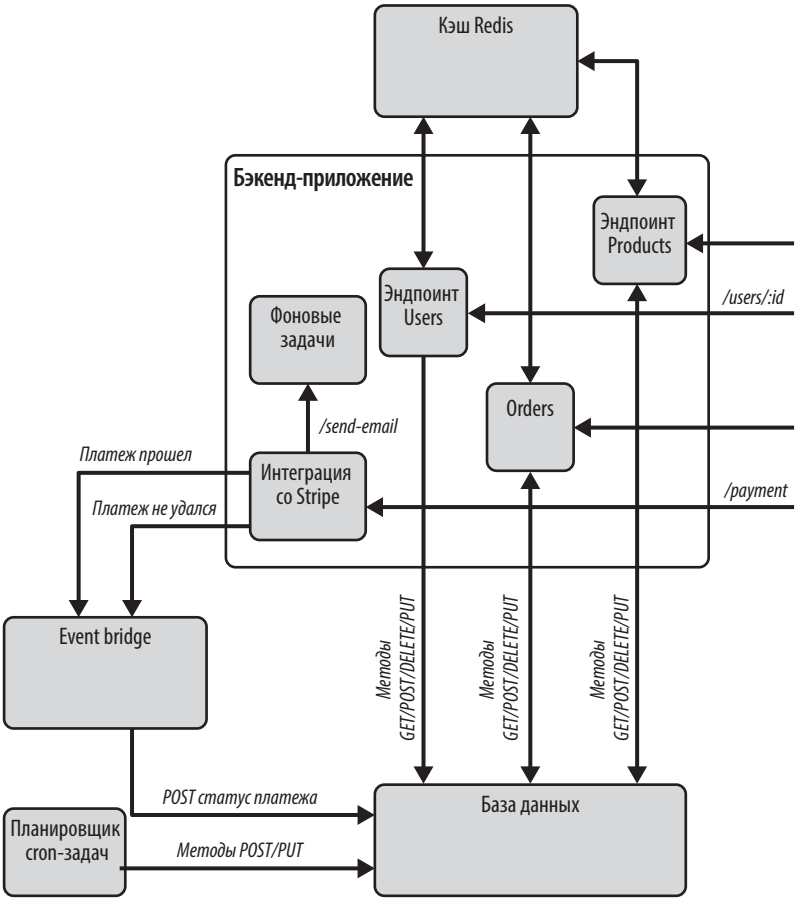

Рис. 10.10. Диаграмма архитектуры с Redis-кэшем

Реализация кэширования с Redis

Чтобы вы понимали, как выглядит код для работы с Redis, вот пример с контроллером товаров в проекте. Сначала нужно настроить приложение для использования кэширования с вашим Redis-сервером. Можно поэкспериментировать с экземпляром Redis в Docker (https://oreil.ly/7mukL). Получив учетные данные Redis, установите несколько пакетов и обновите импорты в файле `app.module.ts`:

```
// app.module.ts
…

@Module({
  imports: [
    AuthModule,
    OrdersModule,
```

```
    StripeModule,
    ProductsModule,
    UsersModule,
    ScheduleModule.forRoot(),
    // делает кэш доступным для всех модулей приложения
    CacheModule.register({
      isGlobal: true,
      store: redisStore,
      host: process.env.REDIS_HOST,
      port: process.env.REDIS_PORT,
      username: process.env.REDIS_USERNAME,
      password: process.env.REDIS_PASSWORD,
    }),
  ],
…
```

Модуль `CacheModule` содержит все необходимые значения для подключения к вашему экземпляру Redis. Полный код можно посмотреть в этом файле (https://oreil.ly/XbVM2): например, `redisStore` является импортируемым пакетом. Все переменные окружения — это значения, которые вы получите из панели управления Redis.

Теперь необходимо обновить эндпоинты, чтобы они действительно использовали этот кэш. Рассмотрим контроллер продуктов и посмотрим, как можно закэшировать ответ для получения всех товаров:

```
// products.controller.ts
…
  // Автоматически кэшировать ответ для этого эндпоинта
  @UseInterceptors(CacheInterceptor)
  @CacheKey('products')
  @CacheTTL(30) // переопределить TTL до 30 секунд
  @Get()
  public async products(): Promise<Array<Product>> {
    return await this.productsService.products({});
  }
…
```

Для полного понимания контекста рекомендуется изучить весь код в этом файле (https://oreil.ly/tASkx), так как здесь используются некоторые импорты. Независимо от применяемого фреймворка, стоит обратить внимание на три ключевых аспекта:

- ответы этого эндпоинта кэшируются автоматически. Это особенно полезно для нагруженных эндпоинтов, чтобы пользователь ощущал повышение скорости;
- необходимо определить ключ кэша. Поскольку кэш хранит данные в виде пар «ключ-значение», этот ключ указывает, где искать данные;
- время жизни кэша (TTL, time-to-live) определяет, как долго значение остается актуальным перед тем, как будет считаться устаревшим.

С кэшированием можно выполнять и более сложные операции, например сохранять ответы сторонних сервисов для ускорения работы. Независимо от фреймворка или даже если бэкенд написан на чистом Node.js, главное, чтобы была возможность реализовать три упомянутых выше пункта, и тогда можно выстроить эффективную стратегию кэширования.

Теперь, когда первая попытка кэширования реализована, стоит взглянуть на улучшение производительности под другим углом.

Продуктовая специфика

Легко увлечься техническими деталями оптимизаций настолько, что вы потеряете из виду исходную проблему. На каждом этапе внедрения улучшений задавайтесь вопросом: как это влияет на продукт? Делает это систему быстрее и удобнее в использовании или здесь требуется баланс компромиссов? Даже если благодаря вашей стратегии кэширования задержки снизятся, насколько вы уверены, что проблемы с кэшем диагностированы и исправлены корректно? Повлияет ли это решение на скорость работы команды в связи с тем, что оно требует поддержки? Это лишь некоторые вопросы, возникающие при оценке оптимизаций со стороны продукта.

Также важно учитывать профессиональную репутацию команды разработчиков и в целом технических специалистов; особенно ярко это проявляется в работе с мониторингом и оповещениями. Следует тщательно выбирать, какие метрики стоит отслеживать, — это повлияет на восприятие вас другими командами. Хотя разработка ПО обычно двигается динамично и упомянутых проблем быть не должно, но то, насколько профессиональной видится команда, влияет на уровень ее автономии и доверия к ней.

Оценивайте производительность с точки зрения пользователя и обязательно тестируйте ее на практике. Можно искусственно ограничить скорость ответов (throttling), чтобы сравнить задержки до и после внедрения кэширования, — это подтвердит ожидаемый эффект. Также продумайте, как решение будет масштабироваться. Выбранный сервис кэширования повлияет на будущие затраты и обслуживание.

Изучите альтернативные методы ускорения приложения: увеличение оперативной памяти сервера для обработки большего числа запросов или настройку программных ограничений на параллельные соединения. Помните: оптимизация требует времени, которое могло бы пойти на разработку нового функционала. Начните с быстрого эксперимента для проверки гипотезы, а затем создайте задачи для исследования долгосрочного решения. Следует определить, с учетом каких критериев будет приниматься решение о необходимости оптимизации: например, отток пользователей из-за медленной работы или отставание по каким-то показателям от конкурентов.

Заключение

В этой главе мы узнали, как находить области для улучшения производительности. Обратитесь к метрикам, которые вы будете использовать для измерения производительности, чтобы понять, что именно требует улучшения. Изучите данные мониторинга: так вы сможете определить, где ваше приложение стабильно выходит за пороговые значения метрик.

Выявив эндпоинты, требующие доработки, выбирайте способ улучшений. Это может быть реализация стратегии кэширования, увеличение ресурсов или изменения в коде. Многие другие стратегии уже относятся к масштабированию, которое мы рассмотрим в следующей главе.

ГЛАВА 11

Вопросы масштабируемости

Мы рассмотрели все аспекты — от архитектуры до логики эндпоинтов, а также вопросы безопасности и отладки. Мы оптимизировали приложение, обеспечив его стабильную работу, и теперь остается подготовить приложение к масштабированию для большего числа пользователей. Это означает, что бэкенд-приложение можно считать готовым. По мере развития продукта вы будете добавлять новый функционал, оптимизировать и вносить изменения, но его основная функциональность уже стабильна.

Масштабирование потребует дополнительных затрат, так как понадобятся более мощные тарифы сторонних сервисов и, возможно, расширение команды. О необходимости масштабирования стоит задуматься, когда другие способы оптимизации производительности уже не помогают справиться с нагрузкой.

В этой главе мы подробнее изучим:

- типы масштабирования;
- стратегии масштабирования.

Решение о масштабировании — серьезная задача, поскольку может потребоваться разделение существующих данных или значительное изменение ресурсов и сервисов. Тщательно изучите доступные варианты и учтите все бизнес-ограничения. Принимаемые решения критически важны для будущей работы систем, поскольку масштабные миграции проводить сложнее.

Типы масштабирования

Существует несколько подходов к масштабированию бэкенд-приложения. *Горизонтальное* масштабирование подразумевает увеличение количества экземпляров, на которых работает приложение. В окружении AWS это может означать добавление дополнительных EC2-инстансов. *Вертикальное* масштабирование — это увеличение мощности или размера ресурса, например добавление ядер CPU или оперативной памяти для одного EC2-инстанса. Также можно комбинировать оба метода. Рассмотрим каждый из них подробнее.

Вертикальное масштабирование

Поскольку вертикальное масштабирование не требует изменений в коде, обычно оно является более простым вариантом, если приложение изначально не разрабатывалось для распределенной работы. Это тот же сервер, но с более мощными ресурсами. Многие компании начинают именно с этого типа масштабирования, так как он позволяет быстро изменить конфигурацию. Кроме того, это, как правило, более бюджетное решение, поскольку оплачивается только один ресурс.

Вертикальное масштабирование имеет свои недостатки. У вас может быть несколько серверов, которые будут масштабироваться вертикально, и это увеличивает затраты и потенциально повышает уязвимость к точкам отказа. Если с сервером что-то происходит, часть приложения может перестать работать без альтернативных ресурсов для запуска. В случае простоя трафик некуда перенаправить. При вертикальном масштабировании (рис. 11.1) также приходится учитывать сложности с обновлениями, которые могут требовать остановки системы.

Рис. 11.1. Пример вертикального масштабирования

Горизонтальное масштабирование

Этот подход (рис. 11.2) обеспечивает повышенную отказоустойчивость и снижает вероятность простоев. Обычно текущие настройки сервера дублируются на нескольких экземплярах. Для распределения трафика между ними потребуется настроить балансировщик нагрузки. Обновления упрощаются, так как трафик можно перенаправлять во время их выполнения. Такое решение обычно более надежно для коммерческих продуктов.

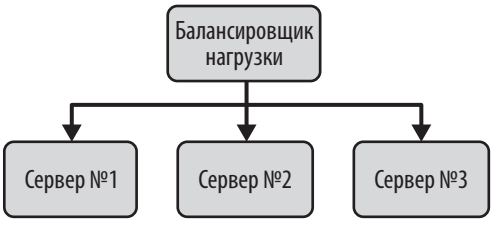

Рис. 11.2. Пример горизонтального масштабирования

Однако реализация горизонтального масштабирования займет больше времени по сравнению с вертикальным. Здесь начинает свою работу команда DevOps или команда SRE (Site Reliability Engineering), которая управляет всеми ресурсами. Это одна из тех областей, где вы сами решаете, насколько глубоко хотите погрузиться в тему. Вы можете в деталях изучить подготовку ресурсов и инструменты масштабирования (например, кластеры Kubernetes (https://oreil.ly/fdpY7), Docker Swarm (https://oreil.ly/CUw99) или Rancher (https://www.rancher.com)) либо ограничиться общим пониманием, что тоже поможет в общении с командой DevOps.

Изучите варианты масштабирования у вашего облачного провайдера

Джефф Грэм дает дополнительные советы по горизонтальному масштабированию.

> Большинство облачных провайдеров предлагает ряд способов автоматического управления горизонтальным масштабированием. Такие сервисы, часто называемые «автомасштабированием» (auto scaling), могут отслеживать трафик или использование ресурсов и добавлять/удалять серверы (или другие ресурсы) по мере необходимости. Это экономит время и затраты — вам реже потребуется управлять кластерами, а неиспользуемые серверы будут автоматически отключаться в периоды низкой нагрузки (например, ночью). Подобные решения обычно доступны и для других облачных сервисов: баз данных, очередей, контейнеров и т.д.

Гибридное масштабирование

Изучая потребности вашей системы, вы можете прийти к выводу, что вам нужен гибридный подход, сочетающий вертикальное и горизонтальное масштабирование. Некоторые экземпляры могут требовать больше ресурсов из-за региона, в котором они размещены. Балансировщики нагрузки помогают избежать простоев, распределяя запросы между различными экземплярами сервера. Если известно, что приложение будет интенсивно использоваться, лучше сразу включить балансировщик нагрузки в первоначальную архитектуру.

Масштабирование ресурсов

Масштабирование касается не только запросов к API. Возможно, потребуется масштабировать ресурсы для обработки задач в очередях (queues). Сигналом к этому служит ситуация, когда сервер получает такой поток запросов или задач, что не успевает их обрабатывать и не укладывается в рамки «стандартного» времени ожидания. Если в очередях остаются необработанные задачи, пора задуматься

о масштабировании. В некоторых случаях задачи следует вынести в отдельный репозиторий и запускать на отдельном сервере, оценив эффективность вертикального масштабирования. Другой ресурс для масштабирования — база данных. Это позволит увеличить пропускную способность данных или поддерживать больше связей.

Необходимо решить, будете вы масштабировать ресурсы вручную или автоматически. Как правило, автомасштабирование — это предпочтительный вариант при работе с облачными сервисами. Важно помнить, что автомасштабирование также называют эластичным масштабированием (elastic scaling), когда ресурсы увеличиваются или уменьшаются в зависимости от количества запросов или событий. Эластичное масштабирование обычно применяется для горизонтального масштабирования, но в редких случаях может использоваться и для вертикального.

Этот подход стоит рассмотреть, если вы знаете, что трафик резко возрастает в определенные периоды года. Такое часто происходит на платформах электронной коммерции и финансовых сервисах во время праздников, когда пользователи совершают больше покупок, чем обычно. Через несколько месяцев трафик возвращается к обычному уровню. Или же сервисы могут столкнуться с резким ростом нагрузки после крупного события. Сайт, который внезапно стал «интернет-звездой», скорее всего, больше выиграет от использования эластичного облака (cloud elasticity), а не от масштабирования, поскольку рост нагрузки краткосрочен. Масштабирование приложения, в свою очередь, предполагает долгосрочное и стабильное использование ресурсов на повышенном уровне, что актуальнее для устойчивого роста, чем для временного всплеска.

Лучшие практики масштабирования

После того как вы определились со стратегией масштабирования, пора приступить к реализации. К этому моменту вы уже должны были провести несколько оптимизаций производительности, чтобы убедиться, что масштабирование действительно необходимо. Ниже приведены шаги, которые помогут вам масштабировать приложения.

Составление плана

Я создала шаблон на основе различных проектов по масштабированию, который можно найти в репозитории проекта (https://oreil.ly/y_P7T). Это хорошая отправная точка, если в вашей компании нет стандартной процедуры масштабирования. Надеюсь, шаблон поможет сформулировать правильные вопросы при составлении плана вместе с другими командами. Один из ключевых вопросов на раннем этапе — оставаться на текущей облачной платформе или мигрировать на другую. До окончательного выбора облачной платформы учитывайте вероятность масштабирования.

Выбор платформы обычно зависит от:

- опыта команды DevOps;
- используемого технологического стека;
- совместимости с внешними сервисами;
- доступного функционала самой платформы.

При масштабировании важно, чтобы выбранный тип легко реализовывался на вашей облачной платформе. Поэтому оцените возможности текущей платформы, прежде чем принимать решение в пользу миграции.

Я работала над проектом, где нам пришлось мигрировать с Heroku на AWS из-за изменений, внесенных Heroku. Это был крайне болезненный процесс, занявший около полутора лет; в нем участвовало несколько команд. В другом проекте мы перешли с GCP на Azure, поскольку материнская организация перевела все свои процессы на продукты Microsoft. На это ушло почти два года. При выборе облачной платформы учитывайте долгосрочную перспективу: миграция, по какой бы причине она ни происходила, — это всегда масштабное мероприятие.

После выбора облачной платформы определите метрики для измерения и мониторинга моментов масштабирования приложений (ключевые метрики рассмотрены в главе 10). Затем распределите зоны ответственности между командами за обработку и тестирование масштабированного приложения. Потребуется:

- план развертывания;
- план отката, гарантирующий, что все команды подготовлены к обновлению, понимают его последствия и имеют механизм отмены изменений при сбоях;
- тестовый план для верификации корректности работы приложения и соответствия новым требованиям к производительности.

Наконец, проведите анализ затрат, чтобы выбранный тип масштабирования не привел к значительным расходам на облачные услуги. В большинстве случаев оптимально горизонтальное автоматизированное масштабирование, но запланированное увеличение ресурсов в определенные периоды может снизить затраты.

Документальное оформление плана

Документируйте детали и принятые решения по выбору стратегии масштабирования. Вопрос о причинах выбора того или иного подхода будет возникать несколько раз в процессе, поэтому держите под рукой диаграмму архитектуры и результаты исследований. Обновите эту диаграмму, чтобы отразить планируемые изменения, как показано на рис. 11.3 (это детализированная версия рис. 11.2).

При распространении обновленной диаграммы архитектуры фиксируйте возникающие вопросы и связанные с ними обсуждения. Это поможет всем отслеживать

ключевые моменты принятия решений и поддерживать организованность процесса. Смело добавляйте комментарии непосредственно в диаграмму — они обеспечат контекст при последующем возврате к обсуждению. Приветствуйте усилия членов команды по совершенствованию документации: это способствует распространению знаний и углубленному пониманию происходящего.

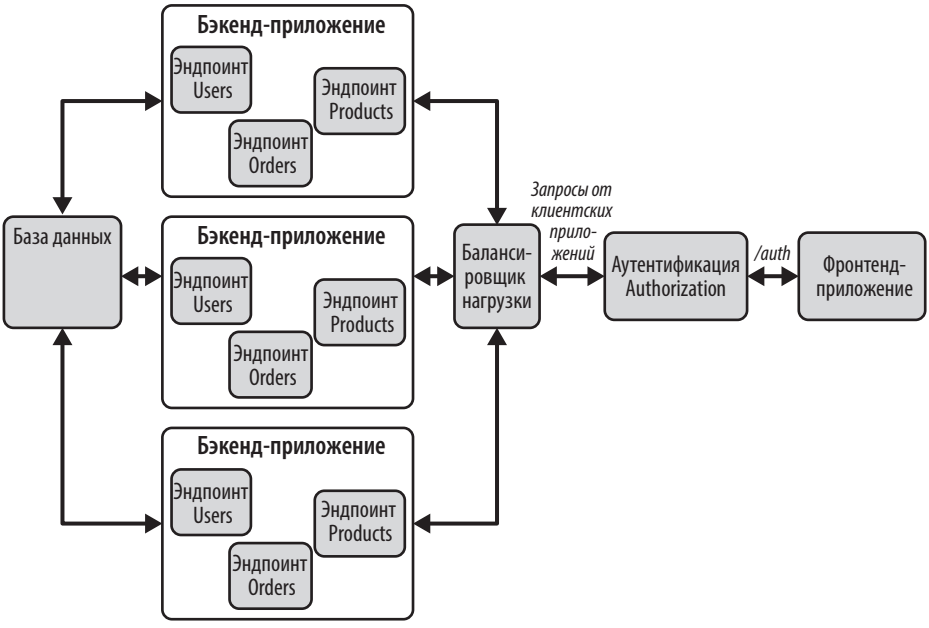

Рис. 11.3. Текущая архитектура с горизонтальным масштабированием

Запуск тестов

Обычно необходимо подготовить автоматизированные тесты, чтобы выполнить первоначальные проверки с новыми ресурсами. Первый раунд тестирования приложения с новым масштабированием следует провести вам вместе с командой разработчиков. Вы обладаете глубокими знаниями о продукте и понимаете, как должен работать код. Это позволяет находить мелкие баги, которые остальные могут не заметить. После программной проверки всех компонентов и подтверждения работоспособности приложения повторно проверьте пайплайн развертывания (deploy pipeline).

Приложение должно работать на ожидаемой версии Node.js без проблем с зависимостями. Также полезно просмотреть логи на наличие ошибок, возникающих при использовании новых ресурсов. Совместно с командой проверьте все эндпоинты, чтобы убедиться в их корректной работе. Здесь особенно важны модульные и сквозные тесты — они сократят объем ручного тестирования. Тем не менее ручное тестирование тоже рекомендуется провести — в первую очередь выбирайте наиболее

нагруженные эндпоинты. Подобное тестирование может занять от нескольких недель до нескольких месяцев в зависимости от сложности системы.

Когда вы внедрили масштабируемое решение, важно провести стресс-тесты, чтобы оценить, как бэкенд-приложение справляется со скачками объемов пользовательских запросов. Такое тестирование позволит убедиться, что ресурсы масштабированы корректно и система не выйдет из строя в экстремальных условиях. Этот тип тестирования дает возможность проанализировать производительность под высокой нагрузкой, проверить способность системы обрабатывать внезапные скачки трафика, оценить ее работу в нестандартных условиях и уровень готовности как самой системы, так и команд к подобным ситуациям.

Стресс-тестирование включает несколько этапов:

- планирование тестирования;
- создание автоматизированных скриптов;
- выполнение скриптов;
- анализ результатов;
- оптимизация и корректировка.

На этапе планирования вы оцениваете текущую производительность системы и определяете цели тестирования. Например: обработка 25 000 запросов в минуту без сбоев, поддержка времени отклика в пределах 3 секунд и корректное отображение сообщений об ошибках. Автоматизированные скрипты должны имитировать действия пользователей, генерирующих эти запросы, а также создавать тестовые данные для использования в запросах.

Затем вы запускаете скрипты и проверяете логи, постепенно увеличивая нагрузку на систему. На этом этапе вы сможете выявить узкие места, где система начинает тормозить. После выполнения всех скриптов и тестирования системы под пиковой нагрузкой проанализируйте результаты, обращая внимание на такие метрики, как время отклика и частота ошибок. Наконец, используйте эти результаты для тонкой настройки системы и оптимизации параметров масштабирования, чтобы достичь оптимального баланса между стоимостью и ожидаемой производительностью.

При переходе на новые ресурсы начинайте с небольших тестов. Убедитесь, что код по-прежнему работает корректно. Вас могут ждать сюрпризы, например, окажется, что при изменении ресурсов важен регистр имен файлов. Поэтому проверять нужно *все*.

Информирование о ходе работ

Информируйте всех заинтересованных о происходящем во время тестов. Вовлекайте другие команды разработчиков, так как им, возможно, предстоит делать то же самое. Рассказывайте о ваших действиях продуктовой команде, чтобы они понимали, почему разработка функционала замедляется в период тестирования.

Сообщайте службе поддержки о любых тестах, затрагивающих пользователей: им нужно быть готовыми к увеличению числа обращений.

Важно учитывать, что у каждой группы в этом процессе есть своя зона ответственности. От команды разработчиков ожидают глубокого понимания работы кода и взаимодействующих с ним систем и механизма их функционирования. DevOps возьмут на себя все изменения инфраструктуры, но у них будут к вам вопросы. Как senior-разработчик, вы можете быть максимально полезны, если будете всегда открыты к диалогу с командой DevOps. Ваша работа часто пересекается с их задачами, особенно в небольших командах.

Если сроки масштабирования значительны, к обсуждениям стоит привлекать middle-разработчиков, чтобы они понимали процессы, скрытые за кодом. Чем больше людей разберется в масштабировании проекта, тем лучше. Так вы распределите ответственность за исправление ошибок и реализацию новых функций. Например, если приложение до сих пор не было контейнеризовано, команде DevOps потребуется ваша помощь, чтобы проверить, насколько корректно оно работает.

Будьте готовы к частым обсуждениям различных вопросов с продуктовой командой и другими сервисами, такими как служба поддержки. Им не нужны технические детали, но важно знать сроки, ожидаемые изменения в производительности и вероятность простоев. Поддерживайте постоянную коммуникацию, чтобы они понимали, что и зачем делается. Это снизит их уровень обеспокоенности и укрепит доверие к вам и вашей команде разработчиков.

Вероятно, потребуется взаимодействие и с другими командами разработчиков, поскольку их код может зависеть от работы вашего приложения. Сообщайте им о планируемых изменениях, чтобы они могли продумать разработку своего функционала. Будет настоящим везением, если именно ваша команда начнет этот процесс. В таком случае вы станете экспертами, к которым другие команды разработчиков будут обращаться за помощью при масштабировании.

Я участвовала в нескольких проектах, проходивших процедуру масштабирования, и ключевым фактором всегда была коммуникация. Технические проблемы возникают постоянно, но именно четкое взаимодействие помогает их преодолевать. Своевременно информируйте коллег о возникающих препятствиях, чтобы общими усилиями минимизировать задержки. Кроме того, настаивайте на дополнительном времени для тестирования, когда это необходимо, — это позволит выявить максимум проблем до перехода в продакшен.

Начало перенаправления трафика

Вместе с командой DevOps настройте автоматическое масштабирование — это оптимально сбалансирует ресурсы, затраты и отказоустойчивость для пользователей. В этой задаче команда DevOps должна взять на себя управление всеми процессами в облачной платформе. И это еще одна область, где вы можете либо

углубить свои знания в области облачной платформы, либо ограничиться необходимым минимумом.

Выбирайте подход к масштабированию как можно раньше. Я работала над проектом, где решение о реализации гибридного масштабирования приняли, когда развитие проекта уже шло полным ходом. Продукт был достаточно зрелым, и его пользовательская база быстро и стабильно росла. На масштабирование ушло около года, и мы столкнулись со множеством сложностей. Масштабирование — это серьезная задача, как техническая, так и нетехническая, решать которую будут различные команды компании. Именно поэтому лучше продумать стратегию на начальных этапах проекта, чтобы избежать подобной ситуации.

Никогда не переключайте весь трафик сразу на только что масштабированные ресурсы — всегда есть риск, что что-то пойдет не так. Постепенный переход — более безопасный метод.

Подготовьте планы отката на случай, если текущая фаза масштабирования пройдет неудачно. В этот период часто оставляют работать и продакшен-сервер, и новые серверы. Так можно минимизировать простои для пользователей во время внесения изменений.

Медленно перенаправляйте трафик на новые ресурсы и наблюдайте за результатами. Внимательно отслеживайте логи, фиксируя новые ошибки или сообщения. Для этого нового трафика полезно настроить отдельное оповещение — так вы быстрее поймете, какую часть продакшена проверять. Лучше выполнять это в непиковые часы, чтобы минимизировать число пользователей, которые столкнутся с возможными проблемами.

После завершения этих мероприятий проведите итоговый разбор со всеми задействованными командами, чтобы убедиться, что внесенные изменения работают корректно.

Заключение

В этой главе мы изучили, как выполнить оптимизацию производительности на уровне архитектуры. Какой бы подход вы ни выбрали, это сложная задача, которая фактически привяжет вас к вашей облачной платформе. Это не страшно, но важно отдавать себе в этом отчет, особенно если вы используете не AWS, GCP или Azure. По моему опыту, миграция на другую облачную платформу — не самая тривиальная задача.

Поддержание коммуникации критически важно. Хотя это не означает, что вы отвечаете за весь процесс, — достаточно следить, чтобы команда разработчиков была открыта к диалогу со всеми, документировать изменения и принятые решения и быть готовыми помочь. Теперь можно сосредоточиться на фронтенде.

ГЛАВА 12

Мониторинг, логирование и обработка инцидентов

Теперь, когда вы построили бэкенд, стоит детальнее изучить инструменты мониторинга и логирования. Именно они предоставят информацию о происходящем в вашем приложении в разных окружениях и помогут принять решения по обработке событий.

Вы уже настроили логирование в коде, определили, какие данные нужно логировать, и имеете достаточный объем для тестирования и проверки. Сейчас цель — извлекать полезную информацию из существующих логов, например выявлять момент возникновения инцидентов и оперативно их устранять.

В этой главе мы рассмотрим:

- различные способы использования логов и данных мониторинга;
- принципы работы различных инструментов;
- планы реагирования на инциденты в продакшене на основе логов и мониторинга.

Мониторинг и логирование нужны не только для выявления проблем. Они рассказывают полную историю о происходящем в вашем фулстек-приложении, включая даже уровень инфраструктуры. Если точно знать, что и когда происходит с приложением, можно понять причины происходящего. Это создает пространство для творчества и позволяет влиять на развитие продукта.

Применение логов и мониторинга

Начнем с определения логирования и мониторинга. В главе 9 мы обсуждали логирование для целей отладки. *Логи* — это сообщения, которые отправляются при возникновении событий в вашем приложении. Обычно они создаются, когда происходит определенное действие, — таким образом сохраняется запись о нем. Изначально логи могут отправляться в текстовый файл где-то в системе, но позже вам надо будет перейти к более надежным инструментам логирования, чтобы упростить поиск сообщений, связанных с конкретными событиями.

Здесь на помощь приходят инструменты *мониторинга*. Они предоставляют данные о ключевых событиях и метриках в реальном времени, позволяя мгновенно обнаруживать ошибки, необычную активность и простои. Мониторинг делает логи более полезными, выделяя наиболее важные действия и сообщения. Это критически важно для приложений, работающих в продакшене и активно используемых. Мониторинг — один из первых способов определить возникновение инцидента. Прежде чем перейти к обработке инцидентов, давайте рассмотрим практическое применение логирования и мониторинга.

Использование логов для отладки

Как известно, логи играют важную роль в отладке. Когда вы сталкиваетесь с ошибкой или хотите проверить выполнение задачи, первым делом стоит поискать соответствующие записи в логах. Быстрый способ локализовать источник ошибки в коде — найти соответствующее сообщение в репозитории. Это приведет вас к конкретной строке кода, откуда отправляется сообщение, и с этого места можно начинать отладку. Также полезно проверить, нет ли других ошибок выше по стеку вызовов (https://oreil.ly/1BqA1). Этот метод помогает быстро найти первопричину, если в первом проверяемом файле ничего не обнаружено.

Иногда ошибки проявляются только в продакшене и не воспроизводятся в стейджинг-среде или локально. Бывает и наоборот — ошибки в стейджинге не возникают в продакшене. В таких случаях стоит проанализировать различия между средами: возможно, проблема связана с нехваткой ресурсов, ограничениями тестовых аккаунтов сторонних сервисов или особенностями тестовой базы данных.

Иногда логи показывают ошибки, которые стабильно возникают в стейджинг-среде, и их нужно исправлять, чтобы сообщения о реальных проблемах оставались читаемыми. В других случаях в логах могут появляться предупреждения, которые, казалось бы, не вызывают проблем. Всегда проверяйте предупреждения, даже если они не приводят к побочным эффектам. Тот факт, что приложение продолжает работать с неопределенной переменной, не означает, что эту проблему можно игнорировать. Предупреждение может указать на первопричину ошибки, но будьте избирательны в том, какие предупреждения и ошибки фиксировать. Иногда они создают информационный шум, из-за которого вы рискуете пропустить даже критически важные ошибки.

Еще один полезный прием — отслеживать ошибки по всему стеку логов. Фронтенд часто дает хорошую отправную точку для поиска проблем, поскольку приложение перестает работать именно там. С помощью логов можно найти ошибки в бэкенде, которые возникли одновременно с ошибкой на фронтенде. Иногда это быстрее приведет к источнику проблемы, чем анализ кода фронтенда. Такой подход экономит время, если корень проблемы все равно кроется в бэкенде.

Логи также должны быть первым местом для проверки при обнаружении несоответствий в работе задач. Проблемы синхронизации данных сложно локализовать,

поскольку они зависят от множества уровней. Анализ логов помогает определить, что следует делать: проверять базу на несогласованность в данных или искать задачи в очереди, которые не выполняются из-за проблем с инфраструктурой. Логи также позволяют проверить, не связаны ли проблемы с данными сторонних сервисов с их простоем.

Еще один полезный прием — добавлять логи в процессе отладки. Иногда ошибку удается локализовать до определенного места, но для полного понимания требуется развернуть изменения в стейджинг-среде. В таких случаях можно добавить дополнительные логи и создать небольшие пул-реквесты, которые дадут больше информации о событиях до или после исходного сообщения в коде. Зачастую можно обернуть функции, возвращающие данные, в логи, где фиксируются переданные параметры и возвращаемые значения. Добавляйте столько логов, сколько нужно для понимания ситуации. Также можно использовать разные уровни логирования (мы обсуждали это в главе 9) вместе с переменными окружения для включения/отключения сообщений.

Как уже упоминалось в предыдущих главах, важно внимательно относиться к передаче данных в логах. Основная причина — сообщения вида `[Object object]` совершенно бесполезны. Если столкнулись с такой проблемой, модифицируйте логирование, используя `JSON.stringify` или аналогичные методы извлечения данных из объектов, поскольку эти сообщения всегда преобразуются в строки.

Логи полезны не только для отладки — они дают наглядное представление о том, как пользователи взаимодействуют с приложением в режиме реального времени. Логи также помогают понять, где требуется расширение ресурсов стейджинг-среды или какие ограничения возникают при работе со сторонними сервисами. Для эффективного использования логов может потребоваться время, поскольку без четкой цели легко утонуть в большом объеме данных. Логи способны ответить на множество вопросов; проявите креативность — придумывайте нетривиальные вопросы и находите на них ответы.

Использование мониторинга для получения информации о своих действиях

Хотя логи предоставляют много информации, мониторинг дает агрегированные данные, на которые можно немедленно реагировать. Мониторинг позволяет настраивать метрики и пороговые значения, чтобы видеть поведение приложения в продакшене или других средах. Вы получите оповещения с метриками о том, как часто вызываются эндпоинты и срабатывают определенные ошибки, а также с прочими метриками, которые вы и различные команды сочтете важными. Такие оповещения обычно дополняются оповещениями об аномальной активности в приложении.

Мониторинг преобразует все ваши логи в полезную информацию. Вы и команда SRE можете принимать решения на основе метрик, получаемых в заданный период времени. Мониторинг работает в реальном времени, позволяя оперативно масштабировать ресурсы при росте нагрузки. Он также помогает быстро выявлять

DDoS-атаки (https://oreil.ly/TCcY7) по количеству и частоте вызовов бэкенда. Зная возможные ошибки, настройте метрику их количества — это поможет разработчикам эффективнее их устранять.

Метрики из глав 10 и 20 предоставляют дополнительные важные данные приложения, которые следует мониторить. Эти метрики показывают изменения времени загрузки страниц, продолжительность ответов API и кумулятивный сдвиг макета (CLS, cumulative layout shift). Система мониторинга формирует графики и статистику на основе этих метрик, для каждой из них устанавливается пороговое значение. При превышении порога срабатывают оповещения, позволяя командам оперативно выявлять и устранять проблемы. Оповещения также гарантируют четкое распределение ответственности за решение инцидентов.

Мониторинг и оповещения упрощают реагирование на инциденты, поскольку команда получает уведомления о превышении порогов. Хотя кажется, что такая система должна быть настроена по умолчанию, мне доводилось работать в проектах, где мониторинг и оповещения внедряли только после нескольких DDoS-атак (https://oreil.ly/TCcY7). Когда они происходили, нам приходилось вручную анализировать логи, чтобы определить перегруженные трафиком эндпоинты, что значительно усложняло процесс. После внедрения системы мониторинга корневые причины проблем становились очевидными, и решение сводилось либо к координации с командой SRE, либо к срочному обновлению кода.

Инструменты мониторинга и логирования

Теперь рассмотрим популярные инструменты логирования и мониторинга, чтобы вы понимали принцип их работы. Я разберу примеры логов и покажу реализацию мониторинга в Datadog (https://oreil.ly/4bK-m) — это один из наиболее распространенных инструментов, которые я встречала в разных организациях. Sentry (https://oreil.ly/cLJN9) и LogRocket (https://logrocket.com) — также отличные продукты, широко используемые в отрасли, и я работала с ними. Я реализую логирование и мониторинг на бэкенде, чтобы вы могли увидеть:

- как работает Datadog;
- как выглядят его дашборды;
- какие функции доступны.

Помните, что хотя некоторые из этих инструментов предлагают бесплатные пробные версии, для продакшен-приложений они платные.

Отмечу, что Sentry можно развернуть на локальном хосте бесплатно, если вы хотите протестировать функционал перед покупкой. Возможно, такой вариант идеально подойдет для ваших задач. Проверьте, поддерживает ли выбранный инструмент мониторинга и логирования локальное развертывание. Все упомянутые в этой главе инструменты предоставляют оба функционала, поскольку логирование и мониторинг тесно связаны.

Datadog предоставляет 14-дневный пробный период для тестирования. В этих примерах мы будем работать с бэкендом, вызывая ошибки в запросах к эндпоинтам и записывая их в логи уровня info. Вам потребуется создать аккаунт и настроить агент (https://oreil.ly/WQ2E4). На рис. 12.1 показан пример вывода после запуска агента на macOS в соответствии с документацией по настройке.

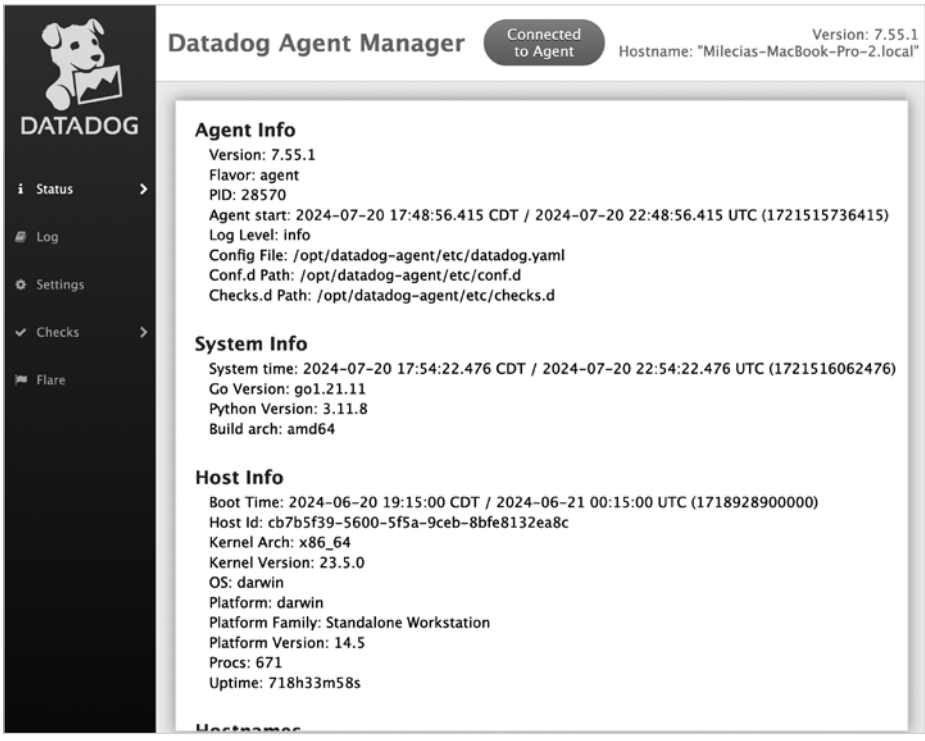

Рис. 12.1. Работающий агент Datadog

Убедитесь, что агент работает, затем установите рекомендуемый в Datadog инструмент для логирования — Winston (https://oreil.ly/J7Dcr):

```
npm i winston
```

Создайте в папке `utils` файл `datadog.ts`, который инициализирует инстанс логгера Datadog с настройками из документации (https://oreil.ly/i3POE):

```
// datadog.ts
import { createLogger, format, transports } from 'winston';

const datadogLogger = createLogger({
  level: 'info',
  exitOnError: false,
  format: format.json(),
```

```
  transports: [new transports.File({ filename: './logs/server.log' })],
});

// Пример логов
datadogLogger.log('info', 'Тестирование логов Datadog...');
datadogLogger.info('Это информационный лог с синим цветом', { color: 'blue' });

export default datadogLogger;
```

Затем вы можете добавить `datadogLogger` в любые контроллеры или сервисы. В этом примере мы добавим его в контроллер заказов следующим образом:

```
// orders.controller.ts
…
import datadogLogger from 'src/utils/loggers/datadog';

@Get()
public async orders(@Param() user: User): Promise<Omit<Order, 'products'>[]> {
  if (!user) {
    datadogLogger.error(`Неавторизованный пользователь: ${user}`);
    throw new HttpException('Unauthorized', HttpStatus.UNAUTHORIZED);
  }
  if (!user.permissions.includes('get:orders')) {
    datadogLogger.error(`Запрещенный пользователь: ${user}`);
    throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
  }

  datadogLogger.info('GET /v1/orders requested');

  try {
    const orders = await this.ordersService.orders();
    datadogLogger.debug(`Заказы: ${orders}`);
    return orders;
  } catch (err) {
    if (err) {
      datadogLogger.error(`Ошибка заказов: ${err}`);
      throw new HttpException('Not found', HttpStatus.NOT_FOUND, { cause: err });
      }
      throw new HttpException('Generic', HttpStatus.BAD_GATEWAY);
  }
}
…
```

Теперь, если перейти в панель управления Datadog и проверить логи, вы сможете видеть все запросы к `GET /orders` и связанные с ними ошибки. По умолчанию на дашборде отображается список логов за последние 15 минут, который в продакшене может содержать тысячи записей, как показано на рис. 12.2.

Так работает один из сценариев мониторинга: вы видите события, которые срабатывают в реальном времени, вместе со статистикой о частоте ошибок, времени их возникновения и их природе. Отслеживая сервисы таким образом, вы можете использовать логи для выявления инцидентов и поиска их первопричин, а также

настроить оповещения, которые срабатывают при превышении заданных пороговых значений в мониторинге.

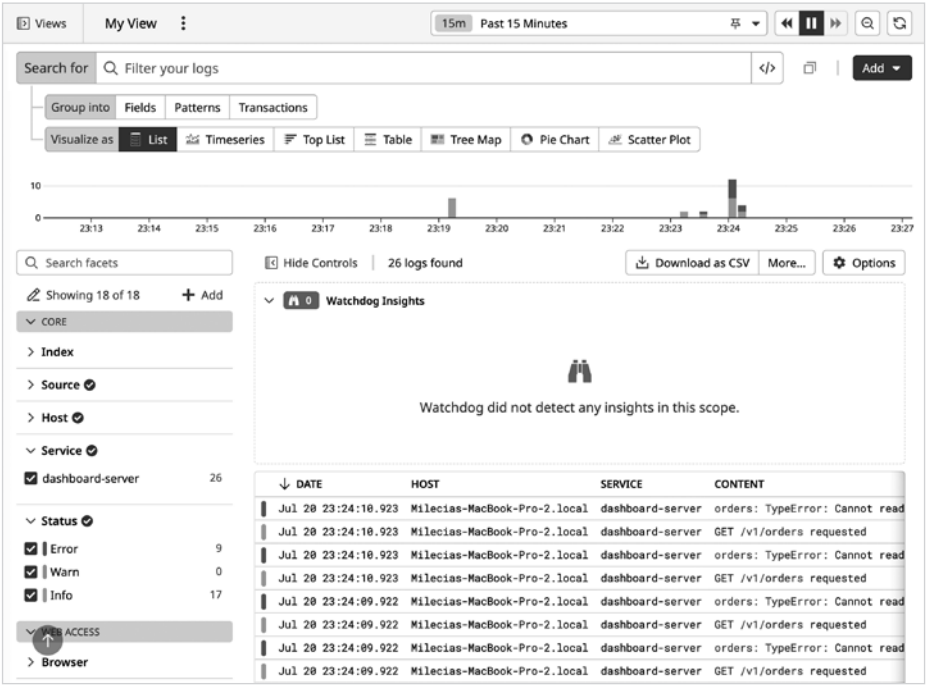

Рис. 12.2. Список логов, отслеживаемых в Datadog в реальном времени

Просматривайте все логи за определенный период времени или фильтруйте их по типу, сервису или другим параметрам. Найдя нужный лог и кликнув по нему, вы увидите детали, как показано на рис. 12.3. Это даст вам информацию, например, о том, что было отправлено в запросе или какая ошибка возникла при формировании ответа. Интересная особенность Datadog — возможность отправлять логи любого типа. Кроме того, инструмент помогает отслеживать статистику частоты вызова пользователями определенных функций.

Datadog предоставляет огромный объем информации, поэтому важно научиться эффективно использовать фильтры. Если вы логируете не только ошибки, на то, чтобы привыкнуть к сортировке всех записей, может потребоваться время. После добавления логов бэкенда на дашборд особенно критичными станут детализированные сообщения в логах — это поможет понять, какие данные использовались для разных запросов и как они связаны с информацией, отправленной с фронтенда.

Важно помнить, что Datadog — лишь один из инструментов, а другие упомянутые ранее решения определенно заслуживают более глубокого изучения. Вполне возможно, что вам больше понравится интерфейс другого инструмента или его функциональность окажется полезнее.

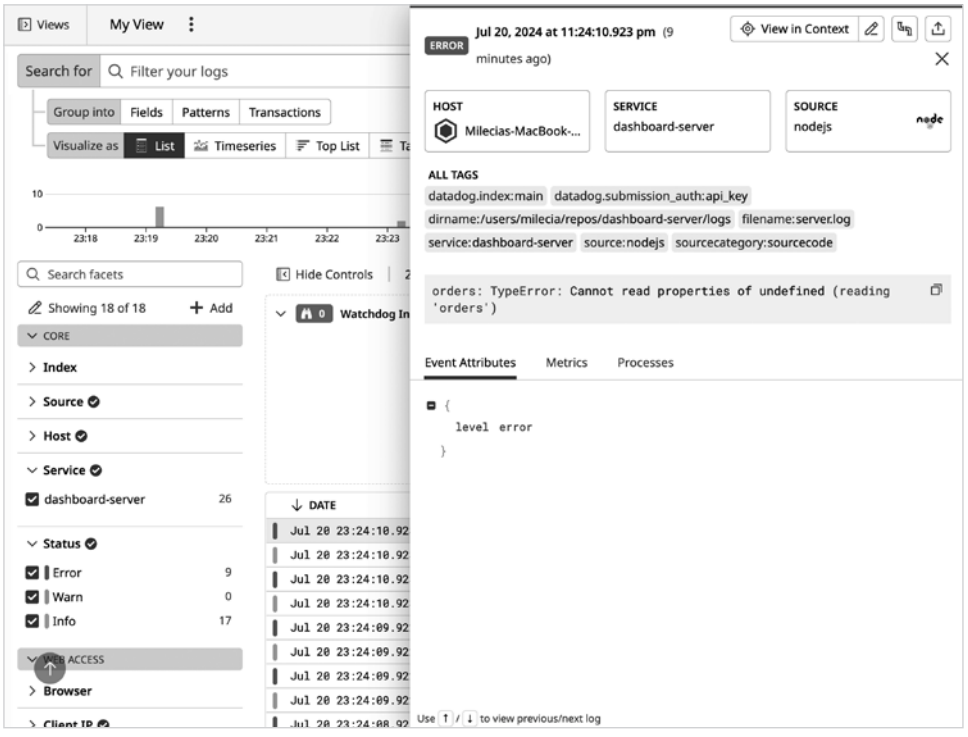

Рис. 12.3. Детализированный просмотр лога в Datadog

Теперь, когда вы понимаете связь между логированием и мониторингом, давайте перейдем к их использованию для обнаружения и обработки инцидентов.

Плейбуки инцидентов

Несмотря на все усилия по поддержанию работоспособности системы в продакшене, приложение всегда может выйти из строя. Пользователи находят неожиданные способы сломать его, сервисы перестают работать, а инструменты дают сбои. Поэтому необходимо заранее подготовить плейбук инцидентов для оперативного и системного решения проблем в продакшене. Разрабатывать такой плейбук следует совместно с командой SRE, поскольку она также будет задействована в процессе.

Этапы плейбука

Плейбук инцидентов включает несколько этапов:

1) обнаружение инцидента;
2) настройка уведомлений по инциденту;

3) определение масштаба и критичности инцидента;
4) уведомление пользователей;
5) уведомление соответствующих команд;
6) делегирование обязанностей по реагированию на инцидент;
7) устранение инцидента.

Рассмотрим каждый из них подробнее.

Этап 1. Обнаружение инцидента

Для начала необходимо четко определить, что считать инцидентом в вашем приложении. Это может быть связано с метриками, которые вы отслеживаете, особенно в контексте нарушений безопасности и простоев. Поэтому важно настроить инструменты логирования и мониторинга так, чтобы они были понятны и привычны всей команде. Помните: невозможно устранить инцидент, если вы не понимаете, что именно произошло. Как только вы составите список возможных инцидентов, при их реальном возникновении команда не будет тратить время на споры о сути проблемы.

Этап 2. Настройка уведомлений по инциденту

После определения типов инцидентов можно приступать к настройке по ним каналов уведомлений. Эффективным решением является создание отдельных каналов в Slack или чатов в Teams для каждого типа инцидентов. В эти каналы следует добавить соответствующих специалистов. Обычно такие каналы небольшие — иногда менее 10 человек. Это делает коммуникацию более целенаправленной, а после выяснения обстоятельств члены группы могут оперативно делиться новой информацией с остальными.

Этап 3. Определение масштаба и критичности инцидента

После формирования соответствующей группы людей в выделенном канале вам потребуется оценить масштаб и критичность инцидента. При анализе степени критичности учитывайте:

- количество пользователей, которые ощутили последствия инцидента;
- перечень недоступных сервисов и задетых функций;
- возможность быстрого исправления;
- объем информации, полученной в ходе первичного исследования проблемы.

Это даст понимание, насколько организация пострадает во время инцидента, а также поможет определить необходимое число сотрудников для работы над инцидентом и ключевые направления их работы. После того как группа реагирования на инциденты оценит масштаб и критичность произошедшего, проинформируйте стейкхолдеров, предоставив им эти данные.

Этап 4. Уведомление пользователей

Начните информировать пользователей о происшествии. Если есть служба поддержки, включите ее в коммуникацию — она общается с клиентами напрямую и должна располагать точной информацией, чтобы знать, как отвечать на вопросы. Уведомив пользователей, сосредоточьтесь на устранении инцидента.

Этап 5. Уведомление соответствующих команд

К этому моменту команды, которые необходимо было задействовать, должны понимать свою роль в устранении инцидента. На основании первоначального анализа следует определить направление работы для всех вовлеченных команд. Как правило, это предполагает совместную работу команд разработчиков и SRE.

Этап 6. Делегирование обязанностей по реагированию на инцидент

Итак, вы находитесь в процессе поиска первопричины. Проведение общей встречи по инциденту — один из самых эффективных подходов из всех, что я наблюдала. Это позволяет:

- оперативно запустить демонстранцию экрана при обнаружении важных деталей;
- повышать эффективность командной работы благодаря оперативному обмену идеями, что предотвращает «топтание на месте»;
- упрощать взаимодействие между несколькими командами: например, разработчики могут программно инициировать действие, а инфраструктурная команда сразу увидит результат, или наоборот.

Такой подход также улучшает восприятие вашей команды и технической команды в целом, что особенно важно в условиях повышенного внимания к инциденту.

Регулярно информируйте стейкхолдеров во время инцидентов

Всегда держите стейкхолдеров в курсе всех обнаруженных деталей инцидента. Поскольку они не обладают такой же технической экспертизой, как вы или другие вовлеченные команды, они могут не до конца понимать происходящее. Это поможет снизить уровень тревожности вокруг инцидента. Каждая организация стремится предоставлять клиентам надежный продукт, поэтому при возникновении проблем естественно ожидать эмоциональных реакций.

Снижайте давление на себя и группу по устранению инцидента, регулярно информируя о выполненных работах. Даже если вы всего лишь сообщаете, что расследование еще продолжается, это лучше полного молчания. Практическое правило, которое я считаю полезным: отправлять обновления каждые 10–15 минут. Это демонстрирует, что вы относитесь к ситуации как к срочной и приоритетной, — так вы укрепите доверие стейкхолдеров ко всем вовлеченным техническим специалистам.

Этап 7. Устранение инцидента

Когда вы сочтете, что нашли решение, проведите тестирование с участием задействованных команд, прежде чем объявлять стейкхолдерам о полном устранении проблемы. Иногда кажется, что все работает, пока не нажмешь другую кнопку. Организуйте многократное тестирование решения максимально разнообразными способами в сжатые сроки. Вы можете информировать стейкхолдеров о работе над решением, но не сообщайте о его готовности до проведения хотя бы базового тестирования. Попросите их воздержаться от коммуникации с пользователями до развертывания исправления в продакшене и дополнительного тестирования там.

Шаблон реагирования на инциденты

Теперь, когда вы знаете основные этапы реагирования на инциденты, вот общий шаблон плейбука на основе плейбуков Google (https://oreil.ly/SDfus), который вы можете использовать. Не забудьте адаптировать его под специфику вашей организации.

- Идентификация (этапы 1 и 2):
- обнаружен инцидент через мониторинг и оповещения:
  - проблема передана соответствующей команде реагирования на инциденты.
- Координирование действий по реагированию на инцидент (этапы 3, 4 и 5):
  - группа реагирования провела приоритизацию инцидента;
  - определены масштаб и критичность инцидента;
  - собраны факты об инциденте;
  - вовлеченные в инцидент команды уведомлены о проведении расследования;
  - стейкхолдеры проинформированы о текущих результатах.
- Устранение (этапы 6 и 7):
  - факты об инциденте исследованы;
  - ключевые стейкхолдеры регулярно получают информацию о действиях по устранению проблемы;
  - приняты меры для минимизации ущерба;
  - устранена корневая проблема;
  - протестированы решения;
  - все затронутые системы и приложения восстановлены и работают нормально.
- Постмортем:
  - проведено ретроспективное совещание;
  - подготовлен отчет с хронологией и перечнем предпринятых действий;
  - выявлены области для улучшения;
  - запланирована доработка процессов или инструментов.

Постмортем без поиска виноватых

Когда вы убедились в том, что инцидент устранен и пользователи уведомлены, наступает время для проведения постмортема (https://oreil.ly/i7tVn). Этот этап так же важен, как и остальные части управления инцидентами, поскольку помогает выявить области для усиления плейбука и повышения устойчивости других процессов. Здесь неуместно искать виноватых или возлагать ответственность на конкретных людей.

Инциденты редко возникают по вине одного человека. Как правило, к ним приводит цепочка событий, и цель постмортема — понять эти взаимосвязи и их роль в возникновении проблемы. Наличие выделенного канала для обработки инцидентов дает преимущество: временные метки позволяют точно оценить продолжительность инцидента и эффективность совместной работы команды над его устранением.

Вот общий алгоритм проведения постмортема.

- Назначьте время и организуйте встречу:
  - избегайте обвинений в чей-либо адрес;
  - сосредоточьтесь на проблемах в процессах, приведших к инциденту.
- Дайте возможность высказаться всем участникам:
  - пусть группа реагирования на инцидент предоставит комментарии о предпринятых действиях;
  - каждый участник — детально опишет свою часть работы;
  - все должны исходить из того, что незначительных деталей нет.
- Составьте официальный отчет:
  - включите в него все выводы;
  - выделите ключевые действия во время инцидента;
  - сформулируйте план улучшений;
  - распространите отчет по организации.
- Поблагодарите участников за правильные действия:
  - если того потребует масштаб инцидента, проведите его межкомандное обсуждение;
  - публично похвалите тех, кто признал ошибки, чтобы способствовать развитию корпоративной культуры.
- Пересмотрите постмортем через несколько дней:
  - позвольте каждому объективно оценить отчет;
  - смоделируйте и проиграйте ключевые шаги инцидента;
  - соберите предложения по улучшению процесса работы над постмортемом.

Проведение постмортемов, где не ищут виноватых, придает уверенности членам команды — они стремятся сообщать о проблемах, а не скрывать их. Если первая мысль сотрудника при обнаружении инцидента — перспектива увольнения, устранение инцидентов может значительно затягиваться. Крайне важно понимать: инцидент никогда не возникает по вине одного человека. В процессе релиза должны быть (или должны были быть) предусмотрены многочисленные проверки. Поэтому анализируйте проблемы процесса, а не людей.

Со временем вы станете ключевым контактным лицом при инцидентах. Так как вы участвуете в настройке базовой функциональности приложения и взаимодействуете со всеми командами, которых касается инцедент, у вас есть целостное понимание системы. Здесь особенно пригодятся составленные вами документы и планы — они позволят быстро находить ответы по интеграциям без погружения в код.

Главное, сохранять хладнокровие и действовать системно. Коллеги могут испытывать стресс из-за проблем в продакшене, и ваша уравновешенность станет ключевым преимуществом. Важно балансировать между эмпатией и контролем эмоций, пока команда решает проблему. Крупные инциденты, например сбой CrowdStrike (https://oreil.ly/w9_zD), могут привести к потерям в миллионы долларов и повлиять на жизненно важные сервисы: здравоохранение, транспорт, финансы. Обязательно обеспечивайте непрерывную коммуникацию всех участников со стейкхолдерами, включая топ-менеджмент.

Заключение

В этой главе мы рассмотрели причины, по которым логирование и мониторинг полезны, а также разобрали некоторые инструменты, которые можно использовать. Ваша команда вместе с другими инженерными командами должна анализировать логи и дашборды системы мониторинга, чтобы находить и устранять проблемы, а также определять первопричины инцидентов. Такой подход позволяет эффективнее, чем при ручной проверке кода и пул-реквестов, выявлять ошибки и понимать, что происходит с вашим приложением. Настройка дашбордов и определение ключевых метрик могут потребовать времени. Необходимо изучить и функционал инструментов, поскольку некоторые из них помогают интерпретировать данные из логов.

Логи и инструменты мониторинга позволяют отслеживать жизненный цикл приложения по мере его развития в продакшене. Убедитесь, что мониторинг настроен корректно, чтобы вы получали оповещения о нестандартных событиях в любом из окружений, например при достижении лимитов ресурсов или аномальной частоте вызовов эндпоинтов. Подобные аномалии могут сигнализировать о возникновении инцидента. Такая система защитит вас, команду и организацию от ситуаций, способных серьезно повлиять на доходы и репутацию.

Часть III

Разработка фронтенда

ГЛАВА 13

Настройка фронтенда

Основная функциональность бэкенда вами уже реализована, поэтому теперь пора сосредоточиться на разработке фронтенда. Именно через него ваши клиенты смогут взаимодействовать с продуктом и использовать весь функционал бэкенда. Фронтенд — это «лицо» продукта вашей компании. Какими бы крутыми ни были ваши данные или быстрыми — API, если ваш продукт неудобен, клиентам не захочется его использовать.

Фронтенд требует иного подхода, чем бэкенд, поскольку здесь необходимо проводить исследования пользовательских предпочтений, чтобы понять, как дизайн и верстка влияют на восприятие продукта. Как правило, продуктовая команда или другие стейкхолдеры напрямую собирают отзывы пользователей, помогают преобразовать их в дизайн-макеты и передают их вам. После этого колдовать начинаете вы.

В этой главе мы рассмотрим:

- как выбрать фронтенд-фреймворк;
- какие дополнительные пакеты могут пригодиться для вашего фреймворка;
- какие долгосрочные решения можно принять сейчас;
- почему в этом фронтенд-проекте будет использоваться React.

Прежде чем настраивать новый фронтенд-репозиторий, изучите имеющиеся дизайн-макеты и ознакомьтесь с дорожной картой по продукту, чтобы понимать, в каком направлении планируется развитие веб-приложения. Задавайте любые возникающие по мере изучения дизайна вопросы, смотрите на него глазами пользователя. Вы удивитесь, сколько пограничных случаев вы сможете предусмотреть!

Как и в любом проекте, нужно с чего-то начать. Прежде чем погружаться в код, проанализируйте доступные фреймворки. Это критически важное решение для жизненного цикла продукта: выбранный фреймворк определит совместимые пакеты и сервисы. Поскольку миграция между фронтенд-фреймворками обычно требует полной переработки приложения, *тщательно* изучите все варианты.

Решения по архитектуре фронтенда

Принимаемые вами решения относительно фронтенда должны быть последовательными, поскольку любые несоответствия будут *заметны* пользователям. Как

и в случае с бэкендом, вам необходимо согласовать соглашения по написанию кода с командой. На фронтенде даже небольшие различия могут напрямую влиять на пользовательский опыт (UX), так как они наглядны и интерактивны.

Ознакомьтесь с дизайнами приложения, которые вы будете разрабатывать, в этом файле Figma (https://oreil.ly/gt0yx). Обратите внимание на места, где потребуются обработка данных и вызовы API, какие части интерфейса можно использовать повторно, а также на любые другие паттерны, которые удастся обнаружить. Это поможет сформировать первоначальное представление о том, как разделить компоненты приложения и какой архитектурный паттерн выбрать.

Перед погружением в проект стоит рассмотреть следующие архитектурные подходы: Model-View-Controller (MVC) (https://oreil.ly/fJ0P7), Model-View-View Model (MVVM) (https://oreil.ly/xpRea), компонентный (https://oreil.ly/qJqTd) и микрофронтенды (https://micro-frontends.org). Ни один из этих подходов не является очевидно лучше других — выбор зависит от целей вашего приложения и используемых инструментов.

В данном проекте мы остановимся на компонентной архитектуре, поскольку она широко применяется в крупных приложениях, а React, который мы используем, изначально ориентирован на компоненты. Дополнительно будем обращаться к методологии атомарного дизайна (https://oreil.ly/fnBe9), но не стоит зацикливаться на терминологии — называйте элементы так, чтобы их структура была понятна команде. Этот подход разбивает фронтенд на минимальные элементы, постепенно собирая из них страницы. Вы встретите его вариации во многих проектах, поэтому стоит с ним ознакомиться — возможно, это подскажет идеи для текущего рефакторинга.

Это отличный вариант для обучения команды. По мере того как вы будете изучать различные архитектуры и принимать решения на основе дизайнов и дорожной карты, старайтесь делиться своими наблюдениями с командой. Для этого вам не нужно готовить презентацию — просто обсудите открытые вкладки и редакторы, которые вы использовали в ходе исследования. Так команда увидит ход ваших мыслей, которые привели к тому или иному решению, и это наблюдение поможет ей выработать собственные критерии.

Во всех проектах, где я работала, ревью пул-реквестов для фронтенда занимало больше времени, чем для бэкенда. Иногда это было даже слишком скрупулезно. Здесь важны даже такие детали, как соглашения об именовании, — переменные передаются между компонентами в разных состояниях, и нужно точно ссылаться на правильные значения. Создайте чек-лист для проверки пул-реквестов, особенно во фронтенде. Вот мой вариант из репозитория фронтенда (https://oreil.ly/pU70I) и пример из документации инженерных практик Google (https://oreil.ly/bJ4tI).

Разработчики приходят и уходят, и технические детали истории развития решений теряются, что может приводить к неожиданным критическим изменениям. Я часто наблюдала, как правки стилей ломали мобильный дизайн, а изменение свойств

(props) в одном компоненте нарушало работу одиннадцати других. Шаблон для ревью пул-реквестов избавит команду от необходимости каждый раз вспоминать, что нужно проверить.

Безопасность во фронтенде критически важна, и мы подробно рассмотрим этот вопрос в главе 19. Сейчас же отметим только, что безопасность станет ключевым фактором при выборе архитектурных паттернов. Например, вам нужно будет выбрать инструменты для валидации вводимых данных (inputs) и пользовательских токенов (user tokens). Также потребуется понимать, как предотвращать распространенные атаки. Эти механизмы будут многократно использоваться в вашем приложении и на протяжении долгого времени влиять на то, как вы создаете формы и выполняете API-запросы.

Что касается создания и организации компонентов, здесь возможны любые решения. Именно поэтому важно строго соблюдать соглашения по коду с самого начала. Знакомо, да?

Вот несколько рекомендаций, которые желательно учитывать при разработке компонентов.

Используйте модульный подход и по возможности отделяйте бизнес-логику от компонентов.

Модульность компонентов упрощает мониторинг различных сфер деятельности компании и позволяет повторно использовать код в приложении. Например, вызовы эндпоинтов можно вынести в отдельные файлы или кастомные хуки (custom hooks), вместо того чтобы обращаться к ним из компонентов напрямую.

Совместно с командой дизайнеров разработайте стандарт для компонентов и терминологии, которую вы будете использовать для обозначения элементов.

Использование единой терминологии поможет всем понимать, какие элементы задействованы и где требуются изменения. В более зрелых организациях команда дизайнеров, скорее всего, уже внедрила дизайн-систему (https://oreil.ly/QNDLe), и это необходимо учитывать.

Реализуйте принципы разделения ответственности (separation of concerns) и единственной ответственности (single responsibility) сразу же, как только начнете разработку.

Разделение ответственности отделяет дизайн от бизнес-логики, что делает компоненты по-настоящему переиспользуемыми. Принцип единственной ответственности предотвращает разрастание кода в компонентах. Если компонент управляет большим числом пропсов (props), вероятно, он берет на себя слишком много функций, что усложняет работу с ним.

Теперь, когда вы определились с архитектурой приложения, создайте архитектурную диаграмму на основе дизайн-макетов. Это поможет разбить дизайн на компоненты и представления и даст возможность их переиспользовать. На рис. 13.1

показан пример каркаса архитектуры на основе макетов (https://oreil.ly/_zOyJ), которые были у вас изначально.

Рис. 13.1. Диаграмма архитектуры проекта

В ходе работы над приложением вы будете добавлять детали и можете указать, где будут выполняться вызовы API и как состояние (state) должно передаваться между компонентами. Эта диаграмма помогает понять, как можно разбить страницы на различные контейнеры и компоненты, которые в них входят. Так вы выявите общие компоненты: таблицы, поисковые строки и кнопки.

После того как вы создадите черновой вариант архитектуры, покажите его команде и попросите высказаться о возможных альтернативных способах декомпозиции приложения. Предложите middle-разработчикам взять на себя проработку контейнеров, добавив детали, такие как состояния, вызовы API и условный рендеринг. Таким образом у вас и команды появляется четкий план, позволяющий приступить к принятию решений по коду.

Выбор фронтенд-фреймворка

Ландшафт фронтенд-разработки меняется постоянно и стремительно. Я помню времена, когда jQuery был максимально приближен к JavaScript-фреймворку. Затем в какой-то момент количество фреймворков резко возросло. Сейчас на выбор доступны React (https://react.dev), Angular (https://angular.io), Vue.js (https://vuejs.org),

Astro (https://astro.build), SolidJS (https://www.solidjs.com), Svelte (https://svelte.dev), Next.js (https://nextjs.org), Remix (https://remix.run) и множество других фреймворков.

Поскольку новые фреймворки появляются постоянно, легко поддаться ажиотажу. Будьте осторожны, выбирая новые фреймворки вместо проверенных. Хотя среди них есть отличные инструменты, не все готовы к продакшену. Некоторые разработчики уже обжигались, вкладывая время в новый инструмент, который затем терял поддержку сообщества. Порой лучший способ опробовать новые фреймворки — использовать их в небольших пет-проектах, а не в полноценных продакшен-приложениях.

Выбрать подходящий фреймворк среди всего многообразия может быть непросто, поэтому вот ключевые ориентиры:

- проанализируйте, поддерживает ли сообщество этот фреймворк и как решаются проблемы на GitHub и Stack Overflow;
- изучите документацию, чтобы понять, насколько быстро в ней можно найти ответы;
- проверьте частоту обновлений фреймворка;
- узнайте, как долго фреймворк существует;
- выясните, какие компании поддерживают фреймворк, выступают спонсорами или владеют им;
- изучите сопутствующие пакеты: инструменты для визуализации данных, для сборки и для управления состояниями, пакеты SDK сторонних сервисов;
- посмотрите опросы разработчиков, например State of JavaScript (https://stateofjs.com) или ежегодный опрос Stack Overflow (https://survey.stackoverflow.co), чтобы понять, какие технологии используются в продакшене и какие тренды существуют;
- проанализируйте вакансии, чтобы узнать, какие технологические стеки используют компании в продакшене;
- уточните, есть ли географические ограничения на разработку фреймворка из-за политической обстановки;
- посмотрите тип лицензии фреймворка;
- убедитесь, что уязвимости исправляются регулярно и оперативно;
- проверьте наличие документации по развертыванию фреймворка на вашей облачной платформе;
- обратите внимание на инструменты, которые используют другие проекты в вашей организации.

Выберите не более пяти фреймворков и определите пять минимальных требований из этого списка. Число пять здесь указано условно — вы можете выбрать больше

или меньше, но такой подход обеспечит достаточную широту и глубину анализа. По мере оценки исключайте фреймворки, которые не соответствуют критериям. Если они не удовлетворяют вашим минимальным требованиям, дальнейшее исследование бессмысленно.

Далее углубитесь в детали фреймворков, используя дополнительные пункты списка. Когда останется два кандидата, подключите команду и выясните, с каким вариантом участники чувствуют себя увереннее. Это может не совпадать с вашими предпочтениями, но приоритетом должно быть удобство сопровождения продукта. Учитывайте экспертизу команды — она сильно повлияет на итоговый выбор. Например, если большинство инженеров работают с React, внедрение Vue или Astro будет сложным, даже если они лучше подходят для продукта.

Использование хорошо «обкатанных» фреймворков — абсолютно нормальная практика. Для продакшен-приложений это часто предпочтительнее, так как гарантирует поддержку при добавлении функционала. Конечно, для совершенно нового проекта (greenfield) хочется выбрать современное решение. В качестве компромисса создайте прототип на стабильном фреймворке и его же аналог — на новом. Пусть команда протестирует оба варианта и сравнит скорость и удобство разработки (DX, developer experience).

В этом проекте будут использоваться React и TypeScript, так как это проверенные инструменты для продакшен-приложений. React соответствует большинству критериев из списка, что делает его надежным вариантом. В то же время, хотя на рынке по-прежнему преобладает React, лучше провести тщательное исследование, чтобы убедиться, что для продакшена вы используете самую актуальную версию фреймворка.

Тот факт, что React сейчас самый популярный фреймворк, еще не означает, что это оптимальный вариант. Другие фреймворки, такие как Svelte и Solid, которые упоминались ранее, предлагают отличный DX и в некоторых случаях даже превосходят React. Я настоятельно рекомендую изучить их и попробовать внедрить эти решения в вашей организации. Наша профессия развивается только тогда, когда кто-то предлагает изменения, а другие видят преимущества этих изменений. Так станьте же тем, кто помогает отрасли двигаться вперед к светлому будущему!

Варианты настройки приложения

Какой бы фреймворк вы ни выбрали, вам потребуется способ запуска фронтенд-приложения. Необходимо выполнить первоначальную настройку, которая определит, как приложение будет собираться, где и как оно сможет работать. На этом этапе стоит рассмотреть инструменты сборки, например Vite (https://vitejs.dev), Rollup (https://rollupjs.org), esbuild (https://esbuild.github.io) и Webpack (https://webpack.js.org) (если без него не обойтись). Выбранный инструмент сборки повлияет на размер бандла, совместимость с другими инструментами, производительность приложения и скорость его развертывания.

Общие компоненты

Каждое фронтенд-приложение включает схожие функции. В вашем проекте будет обработка ошибок, модальные окна, таблицы, формы, поиск, фильтрация и тосты (всплывающие уведомления, toast messages). Посетите любой знакомый сайт и обратите внимание, сколько из этих элементов там используется. Изучите дизайн и начните выявлять общую функциональность на разных страницах. Если у вас есть вопросы по UX, обсуждайте их как можно раньше и чаще. Вносить изменения в проект по мере роста приложения становится сложнее, так как это может привести к непредвиденным побочным эффектам в макетах страниц и нарушить единообразие функциональности.

Выбор пакетов

Здесь приходится идти на компромисс: либо полный контроль над кодом за счет пакетов, разработанных внутри, либо ускорение разработки функционала с помощью готовых решений из npm (https://www.npmjs.com/). Еще одна проблема — раздувание кода из-за пакетов. Это особое искусство, к которому нужно привыкнуть. Вам предстоит решать, когда лучше разработать собственный пакет, а когда использовать существующий, чтобы соблюсти баланс между размером бандла и временем загрузки.

Ниже приведены популярные инструменты для различных задач фронтенд-разработки.

Визуализация данных:

- Chart.js (https://www.chartjs.org);
- D3 (https://d3js.org);
- Three.js (https://threejs.org);
- Recharts (https://recharts.org/en-US);
- VictoryChart (https://formidable.com/open-source/victory/docs/victory-chart);
- Highcharts (https://github.com/highcharts/highcharts-react).

Работа с формами:

- React Final Form (https://final-form.org/react);
- React Hook Form (https://react-hook-form.com);
- Formik (https://formik.org).

Библиотеки компонентов:

- Material UI (https://mui.com/material-ui);
- Chakra UI (https://chakra-ui.com);

- Materialize (https://materializecss.com);
- Semantic UI (https://semantic-ui.com);
- Mantine (https://ui.mantine.dev);
- React Bootstrap (https://react-bootstrap.netlify.app);
- Radix UI (https://www.radix-ui.com).

Управление состоянием:

- Redux (https://redux.js.org);
- MobX (https://mobx.js.org/README.html);
- Zustand (https://github.com/pmndrs/zustand);
- XState (https://xstate.js.org);
- Jotai (https://jotai.org);
- Valtio (https://valtio.dev/docs/introduction/getting-started).

Загрузка данных:

- TanStack Query (https://tanstack.com/query/latest/docs/react/overview);
- SWR (https://swr.vercel.app);
- Apollo Client (https://www.apollographql.com/docs/react/get-started);
- RTK-Query (https://redux-toolkit.js.org/rtk-query/overview);
- React Router (https://reactrouter.com/en/main/guides/data-libs#loading-data).

Стилизация:

- styled-components (https://styled-components.com);
- Emotion (https://emotion.sh/docs/styled);
- CSS Modules (https://github.com/css-modules/css-modules);
- Tailwind CSS (https://tailwindcss.com).

Доступность:

- i18n (https://github.com/mashpie/i18n-node);
- React Aria (https://react-spectrum.adobe.com/react-aria).

Вспомогательные инструменты:

- Lodash (https://lodash.com);
- date-fns (https://date-fns.org);
- Day.js (https://day.js.org);

- jwt-decode (https://github.com/auth0/jwt-decode).

Линтеры и форматтеры:

- ESLint (https://eslint.org);
- Babel (https://babeljs.io);
- Prettier (https://prettier.io);
- js-beautify (https://github.com/beautify-web/js-beautify);
- Biome (https://biomejs.dev);
- dprint (https://dprint.dev/overview).

Менеджеры пакетов:

- npm (https://www.npmjs.com);
- pnpm (https://pnpm.js.org);
- Yarn (https://yarnpkg.com).

Тестирование:

- Jest (https://jestjs.io);
- Vitest (https://vitest.dev);
- React Testing Library (https://testing-library.com/docs/react-testing-library/intro);
- Mocha (https://mochajs.org);
- Cypress (https://cypress.io);
- Puppeteer (https://pptr.dev);
- Playwright (https://playwright.dev).

Теперь, когда у вас есть общее представление о работе приложения и выбранных инструментах, уделите время обдумыванию архитектурного паттерна, который вы будете использовать.

Помните, что эти варианты пакетов — лишь рекомендации на момент написания книги. Экосистема быстро меняется, поэтому важно тщательно все проверить и найти самые современные инструменты, которые используют разработчики. Рано или поздно все инструменты заменяются новыми и, будем надеяться, более совершенными версиями. Поэтому продолжайте изучать, какие решения используют разные команды для своих продакшен-приложений.

Совместная работа с другими командами

По ходу анализа проекта у вас и вашей команды возникнут вопросы по функциональности и пользовательскому опыту (UX). Совместно изучите макеты и создайте

тикеты для каждого экрана. Сформулируйте критерии приемки (https://oreil.ly/PzyJC), описывающие, как должен работать экран. Продумайте, как должны выглядеть поля ввода и обрабатываться ошибки с валидацией. Также учитывайте сценарии, когда пользователь отправляет форму или нажимает кнопку, инициирующую вызов API или UI-событие.

Выявите несколько граничных случаев и добавьте все вопросы в тикеты. Это отличный способ взаимодействия с продуктовой командой: так вы помогаете ей разобраться с технической стороной проекта. В некоторой степени от ваших вопросов будет зависеть то, как продуктовая команда будет рассматривать функционал и другие идеи. В процессе разработки вам предстоит множество обсуждений, но важно провести первоначальное еще до начала работы.

Возможно, вам также потребуется взаимодействовать с командой QA. Даже если ее нет, вам все равно следует подготовить план тестирования. Вместе с QA вам предстоит определить сценарии для тестирования и среды, в которых будут проверяться различные состояния изменений, развернутых в контуре. Новые вопросы можно обсудить с продуктовой командой. В отсутствие QA-специалистов тестирование может стать групповой задачей для команды разработчиков, при решении которой им придется взглянуть на приложение под другим углом.

Обсудите с командой DevOps настройку пайплайнов развертывания для нескольких сред. Участники команды могут заниматься этим параллельно с вашей подготовкой кодовой базы. Вероятно, вам понадобится хранилище для ассетов (например, изображений), поэтому убедитесь, что это учтено в их настройках. Мы рассмотрим некоторые операционные (Ops) задачи, которые вам предстоит решать как фулстек-разработчику, начиная с главы 27, поэтому пока в пайплайны и инструменты интеграции с DevOps-инфраструктурой углубляться не будем.

Заключение

В этой главе мы рассмотрели ключевые аспекты, которые следует учитывать при разработке фронтенда. Специфика новых сложностей заставит вас подходить к деталям реализации иначе, чем в бэкенде. Без предварительного проектирования архитектуры изменения могут «расползаться» и проявляться непредсказуемо. По мере роста приложения вы сможете добавлять в него и адаптировать новые элементы, поэтому не стоит ставить своей целью прямо сейчас предусмотреть все.

Этот проект использует React как один из самых популярных и стабильных фреймворков на момент написания книги, но я рекомендую попробовать и другие варианты. Запустите проект хотя бы еще в паре разных фреймворков, чтобы посмотреть на различия. Несмотря на то что React часто выбирают по умолчанию, другие решения также востребованы и у них есть свои активные сообщества. Сравните их свойства вместе с командой, сделайте это элементом обучения.

ГЛАВА 14

Создание React-приложения

Теперь, когда вы провели исследование и определились с инструментами, пора запустить ваш фронтенд-проект! Вам предстоит настроить React-приложение и все необходимые инструменты для разработки фронтенда.

К этому моменту вы уже тщательно изучили дизайн-макеты, обсудили детали с продуктовой командой и командой разработчиков, а также согласовали с командой DevOps подготовку инфраструктуры для фронтенда. У вас есть схемы и документация, которые помогут вам и команде в создании всех компонентов. Кроме того, вы подготовили достаточно тикетов, которые можно распределить между участниками. Однако при старте проекта с нуля именно вам предстоит инициализировать репозиторий и задать структуру проекта. Это позволит команде параллельно работать над разными частями приложения и быть уверенными, что разработка продвигается без проблем.

В этой главе я расскажу:

- как настроить React-приложение с необходимыми пакетами;
- как разработать первую функцию;
- как написать первый тест.

Не обязательно сразу реализовывать все — достаточно подготовить базу, чтобы команда могла приступить к работе. Возможно, вы займетесь разработкой фоновых компонентов, пока остальные члены команды будут работать над функционалом. При таком подходе вы сможете усилить продуктивность команды: разрабатывая соглашения по коду, документацию и инструменты, вы помогаете коллегам быстрее вникнуть в процесс и эффективнее выполнять задачи.

Настройка начального React-приложения

Первым делом создайте Git в новом репозитории или подключите его к удаленному репозиторию, чтобы система контроля версий была настроена с самого начала. Затем инициализируйте приложение с помощью Vite (https://vitejs.dev/guide). Запустите приведенную ниже команду и следуйте подсказкам. Учтите, что в этом примере указаны подсказки, которые появляются на момент написания книги,

они могут измениться к тому времени, когда вы будете запускать эти команды. Обязательно выберите опции React и TypeScript, когда они появятся.

```
npm create vite@latest

? Project name: › dashboard-web
? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
    Vue
›   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
? Select a variant: › - Use arrow-keys. Return to submit.
›   TypeScript
    TypeScript + SWC
    JavaScript
    JavaScript + SWC
```

Ваш проект называется `dashboard-web`, и при его инициализации будет создано несколько файлов. Одно из удобств Vite — его гибкость. Таким образом, вы можете организовать структуру файлов для компонентов и экранов так, как вам удобно.

Пока что стоит завершить установку и настройку основных инструментов разработчика, которые вам понадобятся.

Настройка линтеров и инструментов форматирования

Пока команда фиксирует и отправляет изменения в условиях жестких дедлайнов, разработчики иногда пишут код, не соответствующий соглашениям, ради экономии времени. Это приводит к появлению нечитаемого и сложного в поддержке кода. Поэтому мы внедряем линтер и инструмент форматирования кода — форматтер. В качестве форматтера используем Prettier (https://oreil.ly/nPbYC), который обеспечит единообразие форматирования. Он проверяет такие аспекты, как пробелы, запятые и длина строк.

Линтером выступит ESLint (https://eslint.org) — он предотвратит появление ошибок: неиспользуемых переменных, глубоко вложенных тернарных операторов или «магических чисел». Эти инструменты обычно используют вместе с husky (https://oreil.ly/nr5fo) для интеграции с git-хуками (https://oreil.ly/ibGXC), например, при коммитах и пушах. ESLint уже настроен при инициализации Vite, поэтому текущий конфигурационный файл можно оставить без изменений, но правила можно уточнить в документации (https://oreil.ly/g52M_). Установите остальные пакеты следующей командой:

```
npm install --save-dev husky prettier
```

Убедитесь, что в конфигурациях Prettier и ESLint нет конфликтующих правил. Я сталкивалась с тем, что это создает проблемы для git-хуков и CI-пайплайнов. Просто обязательно несколько раз запустите команду линтинга, чтобы убедиться, что все проблемы устранены перед коммитом изменений. Хотя мы не будем это реализовывать, но если вам потребуется запускать линтеры для конкретных файлов или в определенном порядке, инструмент lint-staged (https://oreil.ly/8S75r) отлично помогает согласовать работу Prettier и ESLint.

Теперь создайте в корневой папке проекта файл `.prettierrc`. Помните: настраиваемые конфигурации субъективны и отражают предпочтения команды по форматированию. С некоторыми вариантами форматирования все согласятся, с другими — нет, но со временем команда привыкнет. В файл `.prettierrc` можно добавить следующий код (написан с моими предпочтениями):

```
{
    "arrowParens": "always",
    "bracketSpacing": true,
    "trailingComma": "es5",
    "tabWidth": 2,
    "semi": false,
    "singleQuote": true
}
```

Это базовые настройки форматирования для поддержания единообразия кодовой базы — мелочи, которые делают код аккуратным и читабельным. Все доступные опции можно найти в документации Prettier (https://oreil.ly/L6OAB). Также потребуется игнорировать файлы, которые не нужно форматировать (например, `package-lock.json` или конфигурационные файлы). Для этого создайте в корневой папке проекта файл `.prettierignore` с таким содержимым:

```
# Игнорируем артефакты:
build

# Игнорируем все HTML-файлы:
**/*.html

# Игнорируем другие конфигурационные файлы:
**/*.json
.prettierrc
.eslintrc.cjs
src/vite-env.d.ts
vite.config.ts
```

Этот файл работает по аналогии с файлом `.gitignore`. Здесь указываются файлы, которые не должны форматироваться через Prettier. Рекомендуется добавить команду для автоматического форматирования кода по запросу разработчика. Это позволит напрямую редактировать код в файле, чтобы он соответствовал правилам форматирования. Поэтому в файле `package.json` добавьте следующий код в раздел `scripts`:

```
"scripts": {
    …
    "format": "prettier . --write"
  },
```

Также можно предложить разработчикам включить форматирование при сохранении в их редакторах кода. В VS Code эта настройка доступна (https://oreil.ly/IfATs), что избавляет от необходимости вносить правки форматирования в последний момент перед отправкой коммита. Теперь можно настроить husky для выполнения различных команд перед коммитом или отправкой изменений в удаленный репозиторий. Для этого выполните команду:

```
npx husky-init && npm install
```

Проверьте репозиторий — там должен появиться каталог .husky. Он содержит скрипт husky для работы с целевыми git-хуками. Там же находится первый скрипт хука для выполнения действий перед коммитом. Создайте в этом каталоге файл pre-push для размещения команд, выполняемых перед отправкой кода в удаленный репозиторий. Обновите файл pre-commit следующим кодом:

```
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"

npm run lint
npm run format
```

Затем добавьте следующий код в файл pre-push:

```
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"

# npm test
```

Это личное предпочтение, но мне нравится выполнять линтинг и форматирование на этапе перед коммитом (pre-commit), а тестирование оставлять на момент перед отправкой изменений в репозиторий (pre-push). Вы можете использовать любую комбинацию этих действий, если она покажется вам более подходящим вариантом. Уточните, есть ли у кого-то в команде особые предпочтения по поводу правил линтинга и форматирования. Таким вопросом вы уже вовлечете всех в процесс обсуждения первоначальных решений. Проверьте, работают ли ваши конфигурации husky, зафиксировав и отправив текущие изменения. Обратите внимание, что команда test сейчас закомментирована — мы настроим ее чуть позже.

На этом настройка линтеров завершена. Теперь можно переключить внимание на то, как приложение собирается с помощью бандлера Vite.

Настройка конфигурации сборки

Настройка конфигурации сборки крайне важна, поскольку она определяет, как проект будет скомпилирован в код, выполняемый в браузере. Это влияет на:

- способ импорта пакетов и компонентов;
- используемый синтаксис;
- итоговый размер бандлов.

Такие решения обычно принимают исходя из окружений, в которых будет работать приложение. Не все браузеры поддерживают новейшие возможности JavaScript.

Для компиляции TypeScript в JavaScript сначала изучите файл `tsconfig.json`, содержащий параметры `target`, `module` и `lib`. Хотя этот файл часто копируют между проектами без изменений, важно понимать назначение этих параметров — они определяют процесс преобразования TypeScript-кода в JavaScript.

Параметр `target` задает версии ECMAScript (https://oreil.ly/NY9cC), с которыми должно быть совместимо приложение. Это критично, так как этот параметр определяет:

- поддерживаемые браузеры и версии Node.js;
- необходимую версию Node.js на сервере.

Параметр `lib` указывает компилятору, какие возможности JavaScript доступны при выполнении скомпилированного кода. Обычно он совпадает с `target`, поскольку влияет на результат сборки. Исключение — работа в окружении полифилов (https://oreil.ly/gsWX5). Параметр `module` управляет разрешением модулей в приложении. Например, инструкции `import` не работают в старых версиях Node.js. Именно этот параметр определяет, как будут находиться модули в проекте.

Пока можно оставить значения конфигурации без изменений, поскольку значения по умолчанию должны работать в большинстве современных браузеров. Рекомендую бегло ознакомиться с документацией TSConfig (https://oreil.ly/DkYfN), чтобы глубже разобраться в этих параметрах и улучшить понимание процесса сборки.

Выбор значений для `lib`, `target` и `module` существенно влияет на размер бандла и стиль написания кода. Эти параметры определяют, как импортируются функции в разных файлах и как ваш код взаимодействует с другим кодом. Особенно это заметно при создании пакетов, которые используются в других проектах, но эти нюансы важно учитывать в любой разработке.

Когда потребуется собрать весь код в артефакт для развертывания на серверах, настройки сборки можно изменить в файле `vite.config.ts`. Vite использует Rollup (https://rollupjs.org) для создания бандлов, и его стандартных опций обычно бывает достаточно. Если нужно точнее управлять размером бандла или совместимостью с браузерами, в конфигурационный файл Vite можно добавить дополнительные параметры (https://oreil.ly/Zmlc0).

Настройка стилей

Перейдем к стилям. Чтобы реализовать собственную стилизацию и не прибегать к инлайн-стилям, применяйте библиотеку styled-components (https://styled-

components.com). Когда у вас есть компонент для стилей, можно точнее контролировать их поведение: например, обновлять стили через пропсы по условию.

Также мы будем использовать Material UI (MUI) (https://mui.com/material-ui) в качестве библиотеки компонентов. Такая библиотека позволит сосредоточиться на функциональности, так как компоненты уже имеют встроенную доступность и адаптивность. В сочетании со styled-components MUI можно кастомизировать под любую тему. Кроме того, MUI можно расширить, добавив Material Icons (https://oreil.ly/8Yznl), если вам потребуются иконки для кнопок или других визуальных элементов.

Будьте осторожны при использовании библиотек компонентов

Я работала во множестве команд в организациях разного масштаба и ни разу не видела успешной реализации кастомной библиотеки компонентов, если только там не ставили под эту задачу отдельную команду. Обычно все начинается с благих намерений, но со временем кастомную библиотеку без выделенной команды становится поддерживать все сложнее и сложнее, особенно если этой библиотекой пользуются несколько фронтенд-команд с разными требованиями к компонентам.

Итан Браун также замечает на этот счет:

> На мой взгляд, библиотеки компонентов меняются благодаря таким проектам, как Tailwind UI, Headless UI и shadcn/ui. Недостаток классических библиотек в том, что они хороши ровно до тех пор, пока вам не нужно сделать что-то чуть по-другому. Если вы используете MUI и вам требуется реализовать что-то не «по-MUIшному», у вас может уйти вагон времени и вы вконец измучаетесь.
>
> Я считаю, что будущее за такими решениями, как shadcn/ui и Headless UI, где вы просто говорите: «Сгенерируй мне компонент кнопки» — дальше система добавляет его в ваш проект, после чего вы этот компонент полностью контролируете и кастомизируете. Время покажет. Пока же отмечу, что каждая библиотека UI, которую я использовал, включая мою любимую Mantine, со временем становится скорее помехой, чем помощью. Это действительно выбор «из двух зол».

Часто в организации уже есть предпочтения в работе со стилями и компонентами, поскольку команда дизайнеров создает дизайн-систему для единообразия интерфейсов приложений. Поэтому, независимо от используемой библиотеки компонентов, будут ситуации, когда вам придется переопределять существующие стили.

Для начала установите необходимые пакеты с помощью следующей команды:

```
npm install @mui/material @mui/icons-material @emotion/react @emotion/styled
styled-components
```

Теперь вы можете импортировать компоненты MUI и иконки, а также создавать собственные компоненты с помощью styled-components. Следующим шагом инициализируйте тему для вашего приложения. Каждое приложение будет иметь собственные цвета, отступы, шрифты и другие детали. Для подготовки удалите файлы `App.css`, `index.css` и папку assets из директории src. Также необходимо обновить файл `main.tsx`, удалив строку `import './index.css'`. Затем создайте новый файл `theme.tsx` и добавьте следующий код:

```
import { createTheme } from '@mui/material/styles';
import { blue, orange } from '@mui/material/colors';

const theme = createTheme({
  palette: {
    primary: {
      main: blue[900],
    },
    secondary: {
      main: orange[400],
    },
  },
});

export default theme;
```

Этот код задает несколько цветов в объекте темы. Тема определенно будет меняться по мере создания представлений и получения необходимых цветов из макетов Figma, но здесь у вас есть хорошая отправная точка. Рекомендуется ознакомиться с документацией MUI по разработке тем (https://oreil.ly/ygc00), чтобы понять все доступные для настройки параметры, включая темный режим.

Чтобы тема применялась во всем приложении, необходимо добавить `ThemeProvider` в `App.tsx`. *Провайдер* (*provider*) — это распространенный паттерн в React-разработке, который предоставляет компонентам приложения доступ к высокоуровневым функциям и возможностям, таким как темы, модальные окна и управление состоянием. Обычно он реализуется на основе React Context (https://oreil.ly/0LmlB). Откройте файл и обновите его содержимое следующим кодом:

```
import { useState } from 'react';
import { ThemeProvider } from '@mui/material/styles';
import theme from './theme';

function App() {
  const [count, setCount] = useState(0);

  return (
```

```
    <ThemeProvider theme={theme}>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount(count => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.tsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </ThemeProvider>
  );
}

export default App;
```

Вы импортировали только что созданную тему и применили ее через компонент `ThemeProvider`. Теперь все компоненты MUI внутри этого провайдера будут использовать заданную вами тему. Благодаря такому подходу у всех стилей будет единая основа, а это означает, что можно переходить к другим задачам, чтобы подготовить репозиторий для разработчиков.

Настройка тестирования

Тестирование фронтенда, как и бэкенда, поможет избежать регрессионных ошибок во всем приложении. Однако на фронтенде это делать сложнее из-за динамически создаваемых компонентов и выполняемых API-запросов, и эти сложности меняют подход к написанию и проектированию тестов. Поскольку для сборки приложения используется Vite, для модульного тестирования мы будем применять Vitest (https://vitest.dev) и React Testing Library (https://oreil.ly/kRvDD). Установите эти пакеты в проект командой:

```
npm install -D jsdom vitest @testing-library/react
```

Добавьте в `package.json` скрипт для запуска тестов:

```
…
"scripts": {
    …
    "test": "vitest"
}
…
```

Для корректной работы тестов обновите `vite.config.ts`, заменив содержимое файла на:

```
/// <reference types="vitest" />
```

```
/// <reference types="vite/client" />

import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
  }
})
```

Теперь при написании тестов для компонентов все необходимые инструменты будут готовы. Эта конфигурация также интегрируется в CI-пайплайн, а husky запустит тесты через git-хук pre-push. Подробнее о тестировании мы поговорим в главах 21 и 24.

Базовая настройка приложения завершена. Осталось выполнить несколько действий, чтобы его было удобно поддерживать в долгосрочной перспективе.

Настройка файлов CHANGELOG и README

Файлы README и CHANGELOG — это на первый взгляд незначительные файлы, которые на самом деле крайне полезны на протяжении всего жизненного цикла проекта. README можно рассматривать как инструкцию по работе с проектом. В нем следует указать, как настроить проект, перечислить переменные окружения и описать известные особенности. CHANGELOG хранит историю всех изменений в каждом релизе. Все изменения должны фиксироваться, чтобы всегда было понятно, что вошло в каждую версию. Рекомендуется привязывать тикеты к записям в CHANGELOG для удобства дальнейшего обращения.

Альтернативный взгляд на CHANGELOG

В зависимости от релизов в вашей организации могут применяться разные подходы к документированию изменений. Вот мнение Итана Брауна на этот счет.

> Мои взгляды на CHANGELOG изменились. Раньше я использовал подход с простым Markdown-файлом, который обновлялся по мере внесения изменений. Проблема в том, что это часто дублирует работу, уже проделанную в процессе релиза. Обычно релиз включает анализ всех пул-реквестов, объединенных с момента последнего релиза, и составление их краткого описания. CHANGELOG действительно может упростить этот процесс, но он же способен внести путаницу и стать еще одним элементом, требующим поддержки в рамках релиза.

> Я пришел к выводу, что гораздо эффективнее сосредоточиться на документации задач (issues) и пул-реквестов. Если описание каждого пул-реквеста содержит всю необходимую информацию (как и должно быть), составление заметок о релизе значительно упрощается, а усилия по документированию сосредоточиваются там, где это действительно нужно, — на четко определенных задачах и пул-реквестах.

Примеры файлов README (https://oreil.ly/2EPK7) и CHANGELOG (https://oreil.ly/1Or-2) можно найти в репозитории GitHub. Этот вопрос стоит обсудить с командой, поскольку актуальность этих файлов зависит исключительно от их своевременного обновления. Совместно с командой включите эти обновления в стандартный процесс проверки пул-реквестов, чтобы не пропустить их при внесении критических изменений.

Запуск приложения локально

Теперь, когда все инструменты настроены, убедитесь, что приложение нормально работает с вашими файлами конфигураций. Для начала проверьте версию Node, которую используете. Я запускаю это приложение локально на Node 20.10.0 (https://nodejs.org). Если у вас нет возможности переключать версии Node, попробуйте nvm (https://oreil.ly/IguE3), Volta (https://volta.sh) или n (https://github.com/tj/n). После выбора нужной версии в терминале перейдите в директорию проекта и установите все пакеты командой:

```
npm i
```

Затем запустите приложение следующей командой — вывод должен соответствовать указанному ниже:

```
npm run dev

 VITE v5.0.5 ready in 174 ms

  ➜ Local: http://localhost:5173/
  ➜ Network: use --host to expose
  ➜ press h + enter to show help
```

Перейдя по локальному URL-адресу, вы увидите в браузере интерфейс, аналогичный показанному на рис. 14.1.

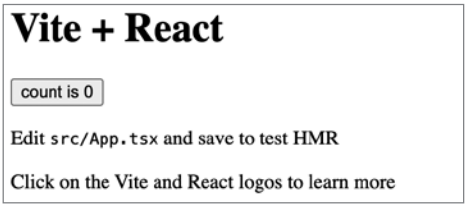

Рис. 14.1. Локально запущенное веб-приложение

Далее необходимо проверить все остальные команды для сборки, тестирования, линтинга и форматирования. Все они должны выполняться без ошибок. Скрипт тестов не запустится, поскольку сейчас нет тестовых файлов. Скрипт сборки должен создать новую директорию dist в корневой директории проекта. Убедитесь, что она указана в `.gitignore`, чтобы не коммитить сборки. Скрипты линтинга и форматирования должны отработать без ошибок.

После успешного выполнения всех проверок следует сохранить текущие изменения — они формируют хорошую основу для приложения. Все, что будет добавлено позже, станет либо новым функционалом, либо улучшением существующих инструментов проекта. Однако сейчас у вас уже есть минимальный каркас для реализации первой функциональности.

Реализация первой функции

Теперь, когда репозиторий настроен и готов к использованию для разработки, можно приступать к добавлению функциональности. Начните с любого элемента дизайна. Я предпочитаю выбирать что-то достаточно простое, но при этом значимое, чтобы сразу выявить потенциальные проблемы. В этом примере мы создадим структуру папок, настроим начальную маршрутизацию и подготовим контейнер для страниц приложения.

Структура проекта

Мы выберем упрощенный подход в стиле React. В директории src создайте две новые папки: components и pages. Папка components будет содержать небольшие переиспользуемые элементы, такие как поисковые строки и общие стилизованные компоненты, а также навигационную панель (navbar) и заголовок. В папке pages разместятся основные модули, формирующие целые страницы, например страницы информации о пользователе и действиях пользователя.

Эту информацию следует добавить в файл README. Она объяснит будущим разработчикам организацию кода, чтобы они могли быстро находить нужные элементы. Когда README будет готов, попросите команду протестировать инструкции и дополнить в случае необходимости.

Добавьте шаблонный код в папку pages для страницы информации о пользователе. Структура папок будет выглядеть так:

```
…
|__ pages
|____ UserInfo
|_______ index.tsx
|_______ UserInfo.Container.tsx
|_______ UserInfo.Container.test.tsx
…
```

Можно разместить следующий код в файле `UserInfo.Container.tsx`:

```
import styled from 'styled-components';

const Container = styled.div`
  background-color: white; // или нужный цвет фона
  height: 100vh;
  width: 100%;
`;

const UserInfo = () => {
  // Запрос информации о пользователе из бэкенда

  return (
    <Container>
      <div>
        <div>Поисковая строка</div>
        <div>Оповещение пользователя</div>
      </div>
    </Container>
  );
};

export default UserInfo;
```

Затем добавьте код в файл `index.tsx` директории UserInfo, чтобы создать модуль для этой папки. Такой файл модуля упрощает импорты и экспорты в приложении, избегая необходимости импортировать модули из отдельных файлов по мере роста проекта. Подробнее о модулях можно узнать в документации MDN (https://oreil.ly/k4jh-) — это важно понимать самому и уметь объяснять команде. Вот код для работы с модулем:

```
import UserInfo from './UserInfo.Container'

export default UserInfo
```

В качестве последнего шага нужно написать хотя бы один тест, чтобы про тестирование не забыли в процессе разработки. Это создаст прецедент наличия тестов в проекте, команда будет понимать важность их написания. В файл `UserInfo.Container.test.tsx` добавьте следующий код:

```
import { afterEach, describe, expect, it, vi } from 'vitest'
import { act, render, screen } from '@testing-library/react'
import UserInfo from '.'

const ui = () => render(<UserInfo />)

describe('<UserInfo />', () => {
  afterEach(() => {
    vi.clearAllMocks()
  })
```

```
  it('должен отображать экран информации о пользователе', async () => {
    ui()

    expect(screen.getByText('Search bar')).toBeDefined()
  })
})
```

Этот подход поможет вам и команде начать реальную разработку, поэтому пока не стоит беспокоиться об идеальном оформлении или полной функциональности кода. Вы будете применять этот шаблон для всех частей приложения — каждый компонент как минимум будет содержать эти три файла с аналогичной структурой.

Настройка маршрутизации

Согласно созданной ранее архитектурной схеме, основной контейнер состоит из двух частей: навигационной панели и текущего экрана. Настроим начальную маршрутизацию для текущего экрана. Для этого потребуется установить React Router (https://oreil.ly/jU8Ao). У него есть несколько альтернатив: TanStack Router (https://oreil.ly/fRkcc) и Next.js (https://nextjs.org). Чтобы установить React Router, выполните команду:

```
npm i react-router-dom
```

Здесь будет определяться, что рендерит приложение в зависимости от URL. Важно помнить: на данном этапе не нужно стремиться к полной функциональности приложения. Цель — создать каркас, позволяющий команде параллельно работать над разными частями. Хотя у вас может возникнуть соблазн углубиться в детали и сделать все самостоятельно, важно вовремя остановиться, чтобы дать коллегам возможность себя проявить.

Теперь создайте в директории src файл `routes.tsx`. В нем будут определены два маршрута для перехода между экранами. Маршрутизация реализуется на раннем этапе, поскольку проекту потребуются уникальные URL для страниц, — это позволит настроить базовую систему переходов. Добавьте в файл следующий код:

```
import { createBrowserRouter } from 'react-router-dom';
import UserInfo from './screens/UserInfo';

const router = createBrowserRouter([
  {
    path: '/',
    element: <UserInfo />,
  },
  {
    path: '/actions',
    element: <div>Actions go burr</div>,
  },
]);

export default router
```

Обновление корневого компонента приложения

Теперь можно обновить `App.tsx` так, чтобы базовая структура интерфейса получила следующий код:

```
import { ThemeProvider } from '@mui/material/styles';
import Box from '@mui/material/Box';
import { RouterProvider } from 'react-router-dom';
import styled from 'styled-components';
import theme from './theme';
import router from './routes';

const AppContainer = styled(Box)`
  display: flex;
`;

const CurrentScreen = styled(Box)`
  width: calc(100% - 240px);
  margin-left: 240px;
`;

const App = () => {
  // Здесь что-то происходит
  return (
    <ThemeProvider theme={theme}>
      <AppContainer component="main">
        <CurrentScreen component="section">
          <RouterProvider router={router} />
        </CurrentScreen>
      </AppContainer>
    </ThemeProvider>
  );
};

export default App;
```

Запустите приложение, чтобы убедиться, что все работает как ожидалось. Вы должны увидеть результат, аналогичный показанному на рис. 14.2.

Рис. 14.2. Каркас приложения с экраном информации о пользователе

Вы можете сверить код этой главы в пул-реквесте (https://oreil.ly/guxuc), чтобы убедиться в правильности настройки. Теперь у вас есть дополнительный код для коммита в репозиторий. Этот пример поможет другим разработчикам начать

реализацию основной функциональности проекта. Первые компоненты дают возможность протестировать новый репозиторий и определить, требуются ли изменения в инструментах или конфигурациях.

Не забывайте делать коммиты часто и небольшими порциями. Это упростит код-ревью и предотвратит потерю большого количества изменений. Рекомендуйте команде придерживаться такого же подхода и зафиксируйте это в соглашениях по пул-реквестам.

Также помните, что на данном этапе проекта вы не обязаны создавать все в одиночку. Именно поэтому эти компоненты, по сути, являются заглушками, а вы пока не углублялись в дизайн. Повторюсь еще раз: важно не зацикливаться на доведении одной части до идеала, пока команда не получит возможность изучить структуру проекта, разобраться в ней, задать вопросы и внести свои предложения. Все будет меняться.

Все ваши скрипты по-прежнему должны выполняться без ошибок. Сейчас подходящее время для тестирования процессов, которые настроила команда DevOps, например развертывания в разных средах. Это позволит команде DevOps как можно раньше устранить возможные проблемы, чтобы разработка шла гладко. Мы подробно разберем настройку CI для фронтенда и бэкенда в главах 25 и 27, поскольку после развертывания инфраструктуры вы с командой, скорее всего, будете заниматься этим больше.

Заключение

В этой главе вы настроили все начальные инструменты и конфигурации для React-приложения. Вы поработали над первыми компонентами, чтобы убедиться, что проект функционирует как ожидалось. Имеющийся код позволит остальной команде приступить к реализации другой функциональности. Хотя ключевые функции, такие как управление состоянием и обработка данных, пока не реализованы, другие разработчики смогут добавить их по ходу работы.

Вы и команда будете устанавливать дополнительные пакеты и настраивать конфигурационные файлы, когда возникнет такая необходимость. По мере углубления в дизайн и функциональность вам потребуется обсуждать детали и, возможно, проводить небольшие демонстрации для сравнения пакетов. Пока репозиторий в текущем начальном состоянии, обязательно проведите встречу, чтобы разобраться, как все работает, — это поможет избежать недопонимания, когда команда начнет работать над разными частями проекта.

ГЛАВА 15

Управление состояниями

Теперь, когда репозиторий создан, а веб-приложение готово к разработке, пора начинать создавать представления на основе макетов Figma (https://oreil.ly/6WcoC) и вашей архитектурной схемы. Контейнер заголовка уже в основном заполнен, поэтому вы можете увидеть, где начинается работа с несколькими аспектами одновременно: состоянием, вызовами API и рендерингом по условию.

Способ управления состояниями определяет, как вы будете обрабатывать динамические данные и значения, изменяющиеся в результате действий пользователя. Существует несколько способов управлять состояниями, и большинство приложений используют их комбинацию, чтобы максимально упростить код.

В этой главе мы поговорим:

- о различных способах управления состояниями и сценариях их применения;
- о жизненном цикле состояния (state lifecycle) в React;
- о том, как делиться состояниями между компонентами;
- об управлении пробрасыванием пропсов (prop drilling).

Понимание того, как работает состояние, когда оно обновляется и почему используются разные подходы, крайне важно. Вам необходимо глубоко разбираться в этом, чтобы обучать членов своей команды и писать максимально оптимизированный код. Если состояниями управляют некорректно, это может привести к странным побочным эффектам (side effects) в пользовательском интерфейсе (UI). Поэтому начнем с изучения принципов работы состояния компонентов.

Как работает состояние компонента

В React есть один метод, который управляет первоначальным рендерингом страницы, — это метод `render`, обычно находящийся в файле `main.tsx`. Вот как он выглядит:

```
ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Аргумент в `createRoot` — это `domNode`, внутри которого управляется объектная модель документа (DOM, Document Object Model). Метод `render` отображает React-компонент, обычно `<App />`.

Однако метод `render` сам по себе не отображает обновления интерфейса при взаимодействии с ним пользователя. Состояние (state) — это механизм React для обработки рендеринга в ответ на действия пользователя: клики по кнопкам, события форм и изменения данных. Как сказано в официальной документации (https://oreil.ly/rRX5Y), состояние можно рассматривать как «память» компонента. После первоначального рендеринга, при обновлении состояния, компоненты перерисовываются.

React использует виртуальный DOM (https://oreil.ly/-bZdl), чтобы сделать повторные рендеры максимально эффективными, обновляя только измененные части страницы. Таким образом, при обновлении состояния перерисовываются только компоненты, где это состояние изменилось. Например, так происходит, когда пользователь нажимает кнопку, выполняется API-запрос для получения данных, которые затем обновляются на странице, или при клике появляется модальное окно.

Здесь используются различные хуки React (https://oreil.ly/LV1N_). *Хук* — это фрагмент кода в рамках React, который позволяет работать с методами жизненного цикла. Хуки можно использовать только в функциональных компонентах React, поскольку они заменили оригинальные методы классов (https://oreil.ly/wt-jN) из старых версий React. Они упрощают совместное использование состояний между компонентами и поддержку компонента на протяжении его жизненного цикла. Некоторые из них — `useState`, `useContext` и `useReducer` — специально предназначены для управления состоянием по мере усложнения проекта.

useState

Этот хук вы, вероятно, будете использовать чаще всего в любом React-приложении. Хук `useState` предоставляет «память» для отдельного компонента. Например, когда пользователь вводит значения в форму, `useState` сохраняет эти значения до их изменения. Состояние компонента изолировано для конкретного экземпляра компонента, поэтому вам не нужно беспокоиться о случайном пересечении состояний. На экране может быть несколько строк поиска для разных частей страницы, например одна в заголовке, а другая в футере, как в этом небольшом примере:

```
import SearchBar from './SearchBar.tsx';

const UserInfo = () => {
  // Получаем информацию о пользователе из бэкенда

  return (
    <Container>
```

```
      <div>
        <header>
          <SearchBar />
        </header>
        <footer>
          <SearchBar />
        </footer>
      </div>
    </Container>
  );
};
```

Каждый компонент `SearchBar` имеет собственное локальное состояние, которое обновляется независимо от другого. Таким образом, любые действия пользователя в первой строке поиска не повлияют на работу второй. Компонент `Container` ничего не знает о том, что происходит в компонентах `SearchBar`, это позволяет им отображаться корректно, поскольку состояния полностью изолированы от него. Если вам нужно состояние, которое каким-то образом влияет на обе строки поиска, следует поднять состояние (raise the state) из компонентов строк поиска в родительский компонент `Container`.

Поднятие состояния (https://oreil.ly/x-EIT) компонента означает перенос состояния из дочернего компонента в родительский. В данном случае это подразумевает перемещение хука `useState` из компонента `SearchBar` в компонент `Container`.

Хук `useState` предоставляет доступ как к значению, так и к методу его обновления. Поэтому состояние обычно объявляют в таком формате:

```
const [userInfo, setUserInfo] = useState<UserInfo>(initialUserInfo)
```

Здесь `userInfo` — текущее значение, а `setUserInfo` — функция для его обновления, которая инициирует повторный рендеринг в React. Параметр `initialUserInfo` задает начальное значение состояния. Хук `useState` идеально подходит для локального состояния компонента, которое не требует совместного использования, например для форм, полей ввода, стилизованных элементов и других часто переиспользуемых частей функционала приложения. Кроме того, этот хук можно использовать как отправную точку для тестирования работы с состоянием перед переходом к более сложным инструментам управления состоянием.

useReducer

Прежде чем внедрять Redux, рассмотрите хук `useReducer` (https://oreil.ly/93rNA), так как принципы их работы схожи. Это встроенный в React способ структурированного представления состояния в компонентах с использованием *действий* (*actions*) и *диспетчеров* (*dispatcher*). В отличие от `useState`, здесь можно задать предопределенные способы обновления состояния вместо единовременного изменения. Например, `useReducer` полезен при работе со взаимосвязанными состояниями — допустим, когда несколько фильтров на странице изменяются в зависимости от выбора других фильтров.

Когда у вас появляется необходимость в подобной функции, это означает, что состояние приложения достигло более высокого уровня сложности, — это важный сигнал. Если вы замечаете, что управление состояниями в компонентах требует множественных изменений в ответ на действия пользователя или что для управления состоянием вы активно передаете пропсы между компонентами, пора задуматься о более эффективном и удобном в обслуживании решении.

Используя `useReducer`, вы можете управлять сложным состоянием как единым объектом вместо управления множеством отдельных переменных. Выбор между `useState` и `useReducer` часто определяется предпочтениями команды. Это отличная возможность, чтобы показать команде различия и совместно принять решение. Сочетание `useReducer` и `useState` хорошо подходит для обработки сложных сценариев по мере роста приложения.

useContext

По мере того как приложение усложняется и растет, состояние неизбежно поднимается вверх по иерархии компонентов. Это приводит к *«бурению пропсов»* (prop drilling), когда значения передаются через слои компонентов, которым они не нужны. Это вызывает неэффективные повторные рендеры и усложняет код. Хук `useContext` (https://oreil.ly/tJoaW) позволяет извлечь состояние и сократить число случаев «бурения пропсов».

Компонент `<ThemeProvider />` — это пример использования контекста (context) для предоставления стилей всем дочерним компонентам в DOM-дереве. Любой компонент, запрашивающий значения из `<ThemeProvider />` с помощью хука `useContext`, получит к ним прямой доступ без необходимости передавать их вниз по дереву контекста.

Подробнее о useContext

Хук `useContext` — мощный инструмент, но у него есть нюансы, на которые обращает внимание Итан Браун. Посмотрите пример в его репозитории на GitHub (https://oreil.ly/BDGhQ). Ниже он делится мнением о влиянии этого хука на производительность:

> Когда состояние поднимается на верхний уровень в крупных приложениях, это часто приводит к проблемам с производительностью, если разработчики не до конца понимают, что может и чего не может делать `useContext`. Если изучить пример в упомянутом выше репозитории, становится ясно, что `useContext` никак не влияет на производительность. Я слишком часто сталкивался с ситуациями, когда кто-то заявлял: «Мы используем этот фрагмент с описанием состояния везде, и мне уже надоело передавать его вниз. Давайте перейдем на `useContext` — это еще и улучшит производительность!»

> Проблема в том, что `useContext` сам по себе не улучшает производительность и даже может серьезно ее ухудшить, если не учитывать, как часто изменяется состояние. Например, тема, язык (для i18n), светлый/темный режим или данные текущего пользователя — все это разумные варианты для `useContext`, так как они меняются редко, а при изменении обычно требуют полного обновления дерева компонентов. Еще один подходящий случай — если вы, например, создаете что-то вроде редактора документов и пользователь долго работает с документом X, а затем переключается на документ Y, тогда `useContext` имеет смысл.
>
> С другой стороны, плохой пример использования `useContext` — настройки сортировки или фильтрации в таблице. Такие параметры меняются часто, и каждое изменение вызовет перерисовку всего дерева компонентов, что негативно скажется на производительности и перевесит любые преимущества `useContext`.

Можно заранее настроить контекст, если известно, что определенное значение будет передаваться в глубоко вложенные дочерние компоненты, хотя чаще это добавляют позже, когда в процессе рефакторинга появляются реальные кейсы использования. Умение вовремя переключиться на другой подход — один из навыков, которые вам пригодятся. Если вы заметили проблему «бурения пропсов», зафиксируйте передаваемые значения и их причину. Затем обсудите с командой возможность перехода на `useContext`. В крупных приложениях на верхнем уровне обычно используется множество провайдеров контекста для управления значениями и состоянием.

Определение уровня приложения для управления состоянием

Когда состояние необходимо совместно использовать между компонентами, следует поднять его выше в иерархии приложения. Например, если на странице есть два фильтра и обновлять ее нужно только после изменения обоих, состояние следует перенести с отдельных фильтров на уровень страницы. Таким образом создается единый источник истины, определяющий момент обновления обоих фильтров.

Если пропсы передаются более чем через 10 уровней, стоит обсудить с командой использование контекста. В крупных приложениях более 10 пропсов могут передаваться через 20 уровней и более, что со временем усложняет поддержку. Если вы начинаете замечать, что пропсы становятся все дальше от родительского компонента, то обращайте на это внимание. Однако передача через несколько уровней — не повод сразу применять контекст. Когда же это начнет существенно мешать разработке, пора будет обсудить варианты.

Фиксируйте наблюдения при работе с кодом и добавлении новых функций. Очень удобно делать заметки в кодовой базе. Не обязательно формализовать процесс

или немедленно организовывать обсуждение с командой. Лично я веду список потенциальных участков для рефакторинга, чтобы не забыть точки роста. На ретроспективах или других командных встречах предлагаю включить их в спринт как технические доработки.

Различные подходы к управлению состоянием

Прежде чем переходить к другим инструментам, я настоятельно рекомендую изучить документацию React (https://oreil.ly/kIlpT) и подробнее разобраться, как работают хуки для управления состоянием. Возможно, вы обнаружите, что встроенных функций вам вполне достаточно. Если же в работе над вашим приложением наступил момент, когда требуется что-то более мощное, чем встроенные хуки, стоит рассмотреть другие инструменты.

При выборе инструментов для управления состоянием вам с командой следует рассмотреть три основных подхода: *reducer*, *atom* и *mutable*.

Подход *reducer* предполагает наличие централизованного источника истины. В этом случае компоненты отправляют действия в центральный источник. Среди преимуществ такого подхода — наличие единого источника и поддержка инструментов разработчика, если вы используете Redux (https://redux.js.org) или Zustand (https://oreil.ly/k6W7c). Однако есть и сложности: необходимо разбираться в новых терминах и концепциях, а инструменты могут работать медленнее встроенных хуков.

Подход *atom* предполагает разделение состояния на мелкие части, которыми можно управлять через хуки. Это похоже на создание состояния в виде переиспользуемых компонентов, поскольку вы можете выводить состояния из небольших частей. В этом подходе часто используют такие библиотеки, как Recoil (https://recoiljs.org) и Jotai (https://jotai.org). Среди преимуществ такого управления состоянием — отличная интеграция с возможностями React и схожий формат работы с хуком `useState`. Основной недостаток заключается в том, что вам может потребоваться представлять состояние в виде графа, а не линейной структуры, что может сбивать с толку junior-разработчиков в вашей команде.

Последний подход, о котором я расскажу, — это подход *mutable*. Он предполагает использование инструментов, которые под капотом применяют прокси для создания изменяемых источников данных, доступных для прямого чтения и записи. Среди библиотек для этого подхода — MobX (https://oreil.ly/p0ElX) и Valtio (https://valtio.pmnd.rs). Полезные особенности этого подхода: гибкость, автоматическое обновление зависимостей при изменении состояния и сокращение количества повторных рендеров благодаря прокси. Недостаток состоит в том, что сложно отследить, когда и как обновляются данные, это усложняет анализ изменений в коде. Кроме того, сочетание изменяемых и неизменяемых данных может снижать ясность кода.

Прокси (proxies) — это просто объекты или функции для других объектов или функций. Это похоже на ситуацию, когда у вас есть эндпоинт, обрабатывающий вызовы сторонних сервисов вместо прямого обращения к ним. В контексте управления состоянием объект прокси отслеживает изменения исходного объекта и запускает функции-обработчики при его обновлении. Подробнее о встроенных в JavaScript возможностях можно узнать в документации MDN по прокси (https://oreil.ly/viUyD).

Один из вышеуказанных подходов обычно используется вместе со встроенными хуками состояния React. Этот вопрос стоит активно обсудить с командой, поскольку принятые решения повлияют на:

- размер бандла;
- опыт пользователей и разработчиков;
- масштабируемость кода, над которым вы работаете.

Оптимальный способ выбрать подход — оценить экспертизу команды и создать несколько небольших прототипов с разными пакетами. Например, можно сравнить управление состоянием с помощью Redux, Jotai и Valtio на простом демо приложении, проведя собственные бенчмарк-тесты. Это покажет:

- скорость работы приложения;
- разницу в размере бандла;
- удобство работы с пакетом в контексте вашего приложения.

Учтите, что различия в производительности сложны и обычно проявляются только в крупных приложениях.

После того как вы совместно с командой завершите этот процесс, можно расширить функциональность компонента `UserInfo.Container`, над которым вы начали работать в главе 14. Для этого проекта среднего уровня сложности мы будем использовать Valtio в качестве инструмента управления состоянием. Хотя это менее популярное решение, оно, в отличие от MobX, поддерживает существующие инструменты разработчика, такие как Redux Toolkit (https://redux-toolkit.js.org). Кроме того, Valtio совместим с Node и Next.js, а не только с React.

Важно пробовать новые подходы, чтобы расширять свой инструментарий. Экспериментирование с незнакомыми технологиями поможет вам быстрее находить решения, когда возникнет подходящий кейс. Не стремитесь стать экспертом во всех инструментах и методологиях — достаточно понимать их возможности, чтобы инициировать обсуждения и вдохновлять коллег.

Настройка менеджера состояний

Это не руководство по внутреннему устройству Valtio, а демонстрация его интеграции в проект. Подробности о функциях и переменных можно найти в документации

Valtio (https://oreil.ly/ZJeCC). Цель раздела — на практическом примере показать, как осваивать новый инструмент и внедрять его в приложение. Для установки выполните команду:

```
npm i valtio
```

Теперь необходимо создать новый файл `UserInfo.State.tsx` в директории src/pages/UserInfo. В этом файле вы определите типы, прокси (https://oreil.ly/AD9Gd), который Valtio будет использовать для отслеживания переменных состояния, а также действия для обновления состояния на основе действий пользователя. Добавьте в этот файл следующий код:

```
import { proxy } from 'valtio';

const OrderStatus = {
  Pending: 'pending',
  Shipped: 'shipped',
  OutOfStock: 'out_of_stock',
} as const;

export type OrderStatusKeys = (typeof OrderStatus)[keyof typeof OrderStatus];

export interface Order {
  id: string;
  productName: string;
  status: OrderStatusKeys;
  orderedDate: string;
  deliveryDate: string;
  totalAmount: number;
  hasBeenReviewed: boolean;
}

export const orderStore = proxy<{
  filter: OrderStatusKeys | undefined;
  orders: Order[];
}>({
  filter: undefined,
  orders: [],
});
```

Важно обратить внимание на `orderStore`. Это прокси, который хранит состояние и обновляет его при изменениях в различных компонентах. Таким образом, он содержит значения, которые обычно задаются в компоненте с помощью хука `useState`. После настройки функциональности и переменных для управления состоянием их нужно использовать в компонентах. Вот фрагмент из файла `UserInfo.Container.tsx`:

```
…
import { useSnapshot } from 'valtio';
import { orderStore } from './UserInfo.State';
…
```

```
const UserInfo = () => {
  const orderSnap = useSnapshot(orderStore);

  const [userInfo, setUserInfo] = useState(initialUserInfo);

  useEffect(() => {
    // Получаем информацию о пользователе и заказы с бэкенда
    setUserInfo(userResponseData);
    orderSnap.orders = orderResponseData;
  }, []);

  return (
    <Container>
      <div>
        <div>{userInfo.name}</div>
        <div>Search bar</div>
        <div>
          {orderSnap.orders.map((order) => (
            <div key={order.id}>{order.productName}</div>
          ))}
        </div>
      </div>
    </Container>
  );
};
…
```

Здесь необходимо импортировать `useSnapshot` и `orderStore`, чтобы работать с состоянием в компоненте. Первое, на что стоит обратить внимание, — переменная `orderSnap`. Именно через нее вы получаете доступ к текущему состоянию в `orderStore`. При первоначальном рендеринге страницы или ее перерисовке из-за изменения состояния вы сможете получить актуальные значения, вызвав `useSnapshot` с `orderStore`.

В хуке `useEffect` видно сочетание локального состояния и состояния из прокси. `orderSnap.orders` — это способ обновления состояния в прокси; данные для него будут поступать из вызова API. В зависимости от компонента и приложения можно использовать такой гибридный подход для максимально эффективного управления состоянием. Здесь проявляется как техническое искусство, так и ваш опыт разработчика. Например, состояние формы можно обрабатывать локально, а другие состояния — через прокси. В основном это зависит от договоренностей в команде, главное, соблюдать выбранный подход последовательно.

В JSX выводимого на экран компонента используется `orderSnap`, чтобы отобразить все заказы из состояния прокси. По мере добавления новых компонентов на страницу (например, фильтрации заказов) вы будете расширять файл `UserInfo.State.tsx` новыми действиями (actions) и применять их во всем приложении. Это очень упрощенный пример использования инструмента управления состоянием. Все это можно реализовать и встроенными хуками React, но теперь вы понимаете, как

система масштабируется вместе с приложением и обрабатывает сложные функции, требующие более глубокого обновления состояния, чем локальный компонент.

Вынеся состояние за пределы компонентов, вы получаете доступ к нему из любой части приложения. В этом сила инструментов управления состоянием и причина их использования в крупных приложениях. Часто не нужно выносить состояние из компонентов, пока приложение не сформировано и в этом не возникает необходимости. Слишком раннее внедрение таких инструментов может замедлить разработку и добавить избыточную сложность.

Заключение

В этой главе мы углубились в детали управления состоянием в React. При разработке приложений встроенных хуков часто бывает достаточно, но при необходимости можно добавить более мощный или универсальный инструмент. Крайне важно обсуждать такие решения с командой, поскольку экспертиза и мнения коллег повлияют на будущую разработку и поддерживаемость приложения (application maintainability).

Для облегчения обсуждения можно создать небольшие демо, где команда увидит работу различных инструментов. Управление состоянием усложняется, когда многочисленные действия обновляют множество состояний в разных компонентах, поэтому согласование общего подхода критически важно. Помните: по мере развития приложения и продукта вам вместе с командой предстоит выбирать оптимальные решения для управления состоянием и тщательно их документировать.

ГЛАВА 16

Управление данными

Вы уже организовали управление состоянием, поэтому теперь пришло время получать и обновлять данные. Поговорите с разработчиками бэкенда, чтобы согласовать ожидания по вызовам API. Как фулстек-разработчик, вы также можете фиксировать необходимые изменения и создавать тикеты для самостоятельной реализации этой работы. На текущий момент вы и разработчики бэкенда уже договорились об ожидаемых эндпоинтах и схеме данных, поэтому можно начинать.

Для повышения производительности рекомендуется ограничивать количество вызовов API, выполняемых фронтендом. Соответственно, вам критически важно понимать, как данные влияют на макет страницы и когда они должны отображаться в сценарии взаимодействия пользователя. Пользователям требуется предсказуемое поведение системы — в противном случае они решат, что с продуктом возникли проблемы или необходимые данные отсутствуют/некорректны. Учет этих факторов поможет определить, как оптимально выстроить потоки данных.

В этой главе мы поговорим про:

- вызовы API;
- асинхронную обработку;
- ситуации, требующие проверки функциональности бэкенда;
- используемые инструменты.

При обмене данными с эндпоинтами существуют определенные нюансы, поскольку необходимо учитывать задержку и сценарии, когда запросы к бэкенду возвращают неожиданный результат. Кроме того, важно продумать, когда следует запрашивать данные и зачем выполняются вызовы. Это позволит определить, возвращает ли бэкенд данные в требуемом формате. Вероятно, по мере сборки интерфейса вам потребуется вносить небольшие корректировки на бэкенде. Прежде чем погружаться в код и запрашивать данные, рекомендую оценить доступные инструменты и обсудить их с командой.

Потенциальные инструменты для получения данных

Экосистема инструментов JavaScript постоянно меняется, поэтому необходимо проводить исследования, чтобы быть в курсе последних инструментов.

В настоящее время стоит упомянуть в обсуждении с другими разработчиками следующие варианты:

- TanStack Query (https://oreil.ly/4CDZJ);
- SWR (https://swr.vercel.app);
- RTK Query (https://oreil.ly/Y7JZf);
- React Router (https://oreil.ly/3zx6h).

Вы обнаружите, что некоторые из этих инструментов хорошо сочетаются друг с другом и идеально работают с Axios (https://oreil.ly/mqrK1) или встроенным Fetch API (https://oreil.ly/hiMro).

Конкретные метрики для оценки библиотек, реализующих получение данных, включают:

- управление кэшем;
- работу с DevTools;
- обработку повторных попыток;
- обработку ошибок;
- поддержку запросов и изменений;
- поддерживаемые протоколы;
- определения API;
- степень интеграции во фронтенд-фреймворк.

Также необходимо учитывать стандартные метрики для пакетов библиотек, такие как:

- размер бандла;
- поддержка сообщества;
- качество документации;
- количество примеров.

Опыт команды поможет принять решение, поскольку каждый разработчик сможет выделить практические аспекты инструментов, с которыми он работал. Если в вашем приложении уже используется Redux, то RTK Query станет отличным вариантом, поскольку это часть набора Redux. Наличие в пакете инструментов, которые «из коробки» реализуют кэширование, мемоизацию или повторное получение данных, может подтолкнуть к выбору полнофункционального решения, такого как TanStack Query или SWR.

Мемоизация — это метод оптимизации, при котором результаты функций, требующих много времени или вычислительных ресурсов, кэшируются. Когда вы передаете функции одинаковые значения аргументов, она возвращает тот же результат. В таких случаях мемоизация возвращает кэшированный результат при передаче тех же аргументов без повторного выполнения функции.

Важно организовать обсуждение и демонстрацию всех предложенных вариантов. Выбранные на этом этапе пакеты существенно повлияют на скорость работы и пользовательский опыт по мере роста приложения, когда его функциональность станет больше зависеть от инструментов обновления данных. Как и с управлением состоянием, инструменты *можно* заменить позже, но это потребует масштабного рефакторинга, который вам и команде придется выполнять постепенно.

Если в бэкенде возникнет необходимость перейти на GraphQL или tRPC (https://trpc.io) — например, при наличии сложных связей данных или глубоко вложенных значений, — ваш фронтенд-инструмент также, вероятно, изменится, чтобы обеспечить совместимость с форматом ответа. В таком случае стоит рассмотреть специализированные решения вроде urql (https://oreil.ly/9kcjK) или Apollo Client (https://oreil.ly/VrF-i). Хотя сравнительные отчеты (https://oreil.ly/g7gJB) по различным инструментам (https://oreil.ly/dAwyt) от независимых экспертов (https://oreil.ly/TKJ0f) доступны публично, рекомендуем провести собственное тестирование производительности хотя бы для трех лучших вариантов, отобранных командой.

После определения инструментов и их комбинаций для получения оптимальных метрик установите и настройте инструменты, согласованные с командой для обработки данных. Затем перейдите к деталям: состояниям загрузки и настройке заголовков. Более глубоко обработку ошибок мы рассмотрим в главе 18, поскольку она включает не только ошибки API.

Обработка вызовов API с помощью Axios и TanStack Query

Для запросов API вы будете использовать Axios, так как он предоставляет множество встроенных конфигураций, а его синтаксис проще, чем у стандартного Fetch API. Для обработки ответов, благодаря богатому функционалу, будет полезен API TanStack Query. Кэширование ответов, пагинация и сокращение количества запросов — лишь часть масштабных задач, которые этот пакет решает из коробки.

Попутно отметим, что Fetch API может выполнять практически те же задачи, что и Axios. Поэтому установка дополнительного пакета может не потребоваться. Однако Axios более оптимизирован с точки зрения разработчика. Это еще один аспект, который следует согласовать с вашей командой, поскольку разные разработчики предпочитают разные инструменты.

Установите Axios, TanStack Query и плагин TanStack Query для помощи в отладке и выявлении потенциальных проблем с помощью следующих команд:

```
npm i @tanstack/react-query axios
npm i -D @tanstack/eslint-plugin-query
```

TanStack Query работает по принципу контекста, поэтому вам потребуется создать клиент запросов — `queryClient` (https://oreil.ly/YjwLI) и обернуть все приложение в провайдера клиента запросов — `QueryClientProvider` (https://oreil.ly/mMogL). В файле `App.tsx` внесите следующие изменения для инициализации и использования обоих компонентов:

```
…
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
…

const queryClient = new QueryClient();
…
const App = () => {
  // Здесь что-то выполняется
  return (
    <QueryClientProvider client={queryClient}>
      <ThemeProvider theme={theme}>
        <AppContainer component="main">
          <NavBar />
          <CurrentScreen component="section">
            <RouterProvider router={router} />
          </CurrentScreen>
        </AppContainer>
      </ThemeProvider>
    </QueryClientProvider>
  )
}
```

Все приложение обернуто в компонент `QueryClientProvider`, что дает ему доступ ко всему содержимому кэша `queryClient`. Вы можете выполнять более сложные операции, например настраивать параметры кэша непосредственно в клиенте запросов. Он также предоставляет множество методов для взаимодействия с кэшем, поэтому обязательно изучите документацию во время первоначальной настройки — это поможет заранее определить необходимые опции. Некоторые из этих параметров естественным образом проявят себя по мере роста приложения и изменения потребностей пользователей.

Прежде чем обновлять компонент `UserInfo` для получения данных через Axios и их сохранения в TanStack Query, выполните одну операцию: в корне проекта создайте новый файл `.env`.

Файл .env

В бэкенде Node.js представил нативную поддержку файлов `.env` в версии 20 (https://oreil.ly/OxTw4). Ранее разработчики использовали пакет `dotenv`. Как и в бэкенде, на фронтенде вы можете применять переменные окружения (`env vars`). Ключевое

преимущество их использования — повышенная безопасность, поскольку они хранятся в памяти, а не на диске. Соответственно, злоумышленнику сложнее получить конфиденциальные данные: они сбрасываются при перезагрузке, а не сохраняются в постоянном хранилище (например, в базе данных), что характерно и для бэкенд-реализаций.

В бэкенде вы можете свободно использовать переменные окружения для передачи секретов или чувствительных конфигураций. Главное — избежать случайной передачи таких данных во фронтенд. На фронтенде переменные окружения имитируются и *никогда* не должны содержать конфиденциальную информацию. При сборке кода они попадают в скомпилированный вывод в виде открытого текста. Поэтому переменные окружения фронтенда подходят только для данных, которые могут быть публичными, — например URL-адресов эндпоинтов или публичных API-ключей.

Это крайне обширная тема, и глубокое погружение требуется лишь при работе с механизмами безопасности или оборудованием. Достаточно запомнить: при использовании переменных окружения секреты обычно не хранятся в репозитории (подробнее см. файл `.gitignore`) и загружаются из CI/CD-пайплайна в зависимости от окружения, куда развертывается приложение.

При локальной разработке или в среде разработки допустимо загружать переменные окружения с диска из `.env`-файла — его значения не влияют на данные продакшена или биллинг сервисов. В зависимости от инструмента сборки приложения доступ к переменным реализуется аналогично бэкенду, например:

```
const apiUrl = process.env.VITE_API_URL;
```

Поскольку в этом примере веб-приложения вы используете Vite на фронтенде, доступ к переменным окружения (https://oreil.ly/-U0gV) будет осуществляться с использованием следующего кода:

```
const apiUrl = import.meta.env.VITE_API_URL
```

Поэтому откройте ваш файл `.env` — именно здесь следует указать URL для вызова вашего API. Причина размещения URL API в этом файле заключается в возможном изменении значения в зависимости от среды, в которую развертывается веб-приложение. Некоторые команды используют отдельную среду для разработки бэкенда, предназначенную для тестирования функциональности как бэкенда, так и фронтенда. Эта среда изолирована от продакшена, поэтому URL API будет отличаться. В упомянутом файле `.env` добавьте строку:

```
VITE_API_URL=http://localhost:3000
```

Для локальной разработки функции оптимально запускать код бэкенда локально в состоянии, которое было описано в главе 11. Такой подход позволяет подключаться к локальному API вместо развернутого и оперативно вносить необходимые правки в бэкенд.

Теперь вы можете создать новый файл `useUserInfo.tsx` в директории src/page/UserInfo. Это кастомный хук (https://oreil.ly/7NSvt), который будет обрабатывать вызовы API за вас. Вы добавите код для выполнения запросов к бэкенду с помощью ранее определенной переменной `VITE_API_URL`. Создание кастомного хука преследует три цели:

- четкое разделение ответственности;
- упрощение тестирования;
- поддержание читабельности кода компонентов.

Добавьте следующий код в ваш кастомный хук:

```
import { useQuery } from '@tanstack/react-query'
import axios from 'axios'

function useUserInfo() {
  const {
    isLoading: ordersAreLoading,
    error: ordersErrors,
    data: orderData,
  } = useQuery({
    queryKey: ['orderData'],
    queryFn: () =>
      axios.get(`${import.meta.env.VITE_API_URL}/v1/orders`)
        .then((res) => res.data),
  })

  const {
    isLoading: userIsLoading,
    error: userErrors,
    data: userData,
  } = useQuery({
    queryKey: ['userData'],
    queryFn: () =>
      axios.get(`${import.meta.env.VITE_API_URL}/v1/users`)
        .then((res) => res.data),
  })

  return {
    ordersAreLoading,
    ordersErrors,
    orderData,
    userIsLoading,
    userErrors,
    userData,
  }
}

export default useUserInfo
```

Важно обратить внимание на вызов хука `useQuery` (https://oreil.ly/eos-F) для получения данных через Axios с кэшированием ответа. Хук `useQuery` предоставляет множество значений для повышения отзывчивости приложения, но сейчас вам нужны только:

- состояния загрузки (`ordersAreLoading`, `userIsLoading`);
- состояния ошибок (`ordersErrors`, `userErrors`);
- данные запросов (`orderData`, `userData`).

В запросах используется определенный вами URL API. Все эти значения возвращаются из вашего кастомного хука в `UserInfo.Container.tsx`.

В файле `UserInfo.Container.tsx` выполните следующие изменения кода для использования этого хука и рендеринга компонента на основе возвращаемых значений:

```
…
const UserInfo = () => {
  const {
    orderData,
    ordersAreLoading,
    ordersErrors,
    userData,
    userIsLoading,
    userErrors,
  } = useUserInfo()

  const { showBoundary } = useErrorBoundary()

  useEffect(() => {
    orderStore.orders = orderData
  }, [orderData])

  useEffect(() => {
    userStore.user = userData
  }, [userData])

  if (userIsLoading || ordersAreLoading)
    return <div>Данные загружаются...</div>

  if (userErrors || ordersErrors)
    return <div>Что-то пошло не так</div>
…

  return (
    <Container component="section">
      <Header
        userName={userStore.user.name}
        joinedDate={userStore.user.joinedDate}
        onSubmitSearch={onSubmitProductSearch}
```

```
      />
      <OrderForm />
      <OrdersTable orders={orderStore.orders} />
    </Container>
  )
}
```

Обратите внимание, как обновляются состояния в хуках `useEffect` на основе их индивидуальных зависимостей. Таким образом, при изменении значений `orderData` или `userData` соответствующее состояние также обновится. Наконец, для состояний загрузки и ошибки предусмотрены два компонента с условным рендерингом. Это улучшает UX, чтобы пользователи не гадали, почему они не могут взаимодействовать со страницей.

Обработка состояний загрузки

Вам следует создать отдельные компоненты для состояний загрузки. Для каждого компонента на странице, который динамически получает данные, можно реализовать разные состояния загрузки. Здесь поможет ваша библиотека компонентов. Обсудите это с командами дизайнеров и продуктовой командой. MUI предлагает различные компоненты для состояний загрузки (https://oreil.ly/xnv_f), которые можно адаптировать под ваше приложение, — это станет отправной точкой для обсуждения.

Например, поиск заказов в таблице на странице информации о пользователе и загрузка общих данных пользователя требуют отдельных вызовов API. Теоретически можно заблокировать всю страницу до получения всех ответов, но это негативно скажется на UX: пользователи не смогут взаимодействовать с интерфейсом. Поскольку один ответ, вероятно, придет раньше другого, для каждого случая потребуется отображать немного отличающееся состояние загрузки.

В качестве простого примера: можно реализовать два разных компонента загрузки — для шапки пользователя и таблицы заказов. Они будут условно рендериться, что заменит единую проверку обоих состояний загрузки, как показано в следующем примере и на рис. 16.1:

```
// UserInfo.Container.tsx
…
  return (
    <Container>
      {userIsLoading ? (
        <CircularProgress />
      ) : (
        <Header
          userName={userStore.user.name}
          joinedDate={userStore.user.joinedDate}
          onSubmitSearch={onSubmitProductSearch}
        />
      )}
```

```
      <div>Панель поиска</div>
      {ordersAreLoading ? (
        <CircularProgress color="secondary" />
      ) : (
        orderSnap.orders.map((order) => (
          <div key={order.id}>{order.productName}</div>
        ))
      )}
    </Container>
  )
…
```

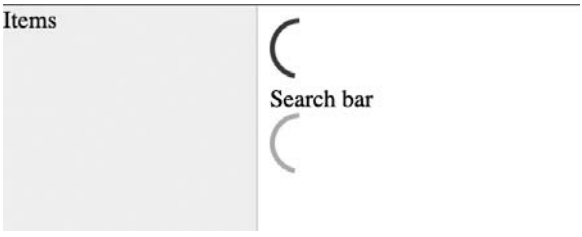

Рис. 16.1. Скриншот нескольких компонентов загрузки в коде

При таком подходе вы не используете общий компонент (который возвращался бы только при загрузке обоих запросов данных), и у вас появляется больше гибкости в управлении отображением для пользователей. Состояния загрузки помогают работать с асинхронной природой вызовов API во фронтенде. Для обработки данных можно:

- объединять промисы (https://oreil.ly/P3SN3);
- использовать `Promise.all` (https://oreil.ly/L34oM);
- вручную создавать состояния загрузки.

TanStack Query автоматически управляет этим отображением через возвращаемое значение `isLoading` до получения ответа. При необходимости несколько запросов можно объединить, настроив параметры TanStack Query.

Обработка состояний ошибок

Мы подробнее рассмотрим обработку ошибок в главе 18, так как это обширная тема, но уже сейчас можно дополнить ваш компонент базовой реализацией.

Пока воспринимайте эту реализацию как временное решение, которое нужно, чтобы не тормозить работу команды. При возникновении ошибки в любом из API-запросов важно обработать ее корректно — это предотвратит аварийное завершение работы приложения для пользователя. Простое сообщение об ошибке, информирующее пользователя о проблеме, лучше, чем его отсутствие:

```
…
  useEffect(() => {
    // Сохраняем данные заказов в хранилище
    orderStore.orders = orderData;
  }, [orderData]);

  useEffect(() => {
    // Сохраняем данные пользователя в хранилище
    userStore.user = userData;
  }, [userData]);

  if (userErrors || ordersErrors)
    return (
      <Alert icon={<CheckIcon fontSize="inherit" />} severity="error">
        Что-то пошло не так
      </Alert>
    )
…
```

Теперь пользователи будут видеть это сообщение при ошибках в любом из запросов. В главе 18 вы узнаете, как разделить их на отдельные сообщения для разных уровней ошибок и управлять ими в различных компонентах.

Настройка заголовков запроса

Как вы увидели из этого примера, вы подобрались к той точке развития приложения, когда потребуется выполнять множество запросов для реализации функциональности, ожидаемой пользователем. Это подразумевает отправку запросов к бэкенду с определенными значениями в заголовках, такими как токены доступа или ожидаемые типы данных. Соответственно, вам необходимо настроить инструменты получения данных для обработки заголовков, чтобы избежать таких проблем, как ошибки CORS (https://oreil.ly/Z3Ztj) или отправка данных в некорректных форматах. Вы можете настроить заголовки для каждого запроса отдельно или же централизованно на уровне приложения, чтобы унифицировать все запросы.

Предположим, что вы с командой приняли решение настроить для этого приложения заголовки через Axios (https://oreil.ly/ucot2) на глобальном уровне, — это повысит удобство сопровождения и упростит отладку. Хотя заголовки можно настраивать и для шаблонов эндпоинтов, авторизация остается одним из наиболее распространенных вариантов глобальной конфигурации. Для реализации создайте файл `axios.config.ts` в директории src. В нем определите базовый URL для всех запросов, тип содержимого и параметры авторизации:

```
import axios from 'axios'

const AUTH_TOKEN = localStorage.getItem('dashboard_web_auth')

axios.defaults.baseURL = import.meta.env.VITE_API_URL
```

```
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN
axios.defaults.headers.post['Content-Type'] = 'application/json'
```

Пока что `AUTH_TOKEN` — это значение-заглушка, которое может (но не обязано) браться из `localStorage`. Этот аспект мы подробно разберем в главе 19 при обсуждении вопросов безопасности. Поскольку базовый URL задан глобально, он будет применяться для всех запросов в веб-приложении. Если вам потребуется отправлять запросы на разные базовые URL, вы можете:

- указывать их индивидуально для каждого запроса;
- использовать отдельные инстансы Axios.

После обсуждения с командой определитесь с подходами: например, создать хуки для запросов или разработать SDK для бэкенда. *SDK* (Software Development Kit) — это пакет, экспортирующий:

- методы для выполнения запросов к API;
- типы для методов и параметров;
- документацию методов.

Такой подход целесообразен при использовании API в нескольких фронтенд-приложениях или других сервисах бэкенда. Примерами популярных SDK служат SendGrid (https://oreil.ly/vqiOo) и Google Maps (https://oreil.ly/ND8MA).

Также необходимо согласовать стратегию кэширования для веб-приложения — этому посвящена глава 20. Настраивая остальные фронтенд-запросы, убедитесь, что фронтенд не дублирует функциональность бэкенда.

Когда проверять функциональность бэкенда

Каждый раз, когда вы начинаете выполнять вычисления на фронтенде, стоит внимательнее изучить бэкенд. Хотя фронтенд способен делать вычисления, их выполнение на бэкенде обеспечивает бóльшую согласованность между браузерами и клиентскими устройствами. Поэтому, столкнувшись с необходимостью вычислений для данных на фронтенде, проверьте, не целесообразнее ли перенести эту логику на бэкенд. Так вы сможете выполнить все расчеты на сервере всего один раз вместо тысяч повторений на клиентах либо закэшировать результат и отправить его в клиента.

Если вы активно форматируете данные на фронтенде, убедитесь, что для форматирования есть причины. Когда причиной являются запросы с других сервисов, использующих этот эндпоинт, рассмотрите возможность создания отдельного эндпоинта. Разделите бизнес-требования, чтобы пользовательские функции не страдали от изменений, предназначенных для других сервисов.

Еще один тревожный сигнал — получение фронтендом больших объемов данных. Это может происходить при пагинации, сортировке или фильтрации, если они не были учтены в исходных спецификациях API. Обсудите это с командой: вероятно, такие функции потребуется добавить на бэкенд для нескольких эндпоинтов. Унификация формата запросов данных для разных страниц улучшит качество кода по мере роста приложения — обязательно детально проработайте этот вопрос с командой.

Стоит также обратить внимание на ситуации, когда для получения данных требуется выполнять несколько запросов с последующим объединением результатов на фронтенде. Обсудите и этот вопрос с командой, чтобы определиться: возможно ли создать для этих данных единый эндпоинт или действительно необходимо разделить эндпоинты. Выполнение множественных запросов не всегда плохо, если это оправданно для долгосрочной разработки. Основанием для разделения эндпоинтов может стать оптимизация, например соблюдение принципа разделения ответственности на бэкенде.

Заключение

В этой главе мы углубили знания об обработке запросов данных и ответов с помощью инструментов, упрощающих управление этим процессом. Кэширование на фронтенде — масштабная задача, поэтому наличие пакета, который поддерживает его «из коробки» и не требует сложного конфигурирования, будет крайне полезно. Инструменты, автоматически сообщающие статус запроса, помогают показывать пользователю понятные сообщения на странице. Если эти инструменты сочетаются с инструментами управления состоянием, то вместе они формируют значительную часть фронтенда.

Внедрение последовательных и продуманных стратегий управления данными на этом этапе обеспечит безопасность кода в будущем. Базовый функционал реализован, и ваша команда готова активно развивать конкретные компоненты приложения. По мере насыщения продукта функционалом некоторые решения потребуют обновления, добавления или удаления. Именно на этом этапе следует ужесточить требования к код-ревью, чтобы сохранить высокое качество кода.

ГЛАВА 17

Кастомные стили

У вас уже есть дизайн-макеты для создания первоначальной версии приложения, но они пройдут множество итераций по мере разработки функционала и получения продуктовой командой обратной связи от пользователей.

Вы будете постоянно взаимодействовать с командой дизайнеров, чтобы стили приложения сохраняли единообразие по мере добавления новых функций и включения новых специалистов в их команду.

Шрифты, размеры, отступы, цвета и кастомные компоненты (например, степперы, модальные окна и контейнеры) будут многократно использоваться в приложении.

Иногда в работе команды дизайнеров может возникать несогласованность из-за большого объема задач. В таких случаях сразу озвучивайте проблему, чтобы совместно выбрать оптимальное как для пользователей, так и для разработчиков решение. Если вы заметите, что дизайн снижает доступность, также обязательно сообщайте об этом.

В этой главе мы рассмотрим:

- доступность;
- единообразие в дизайнах;
- пользовательские темы;
- адаптивный дизайн.

Вам придется опираться на свои ощущения и опыт, чтобы распознавать, когда дизайны в кодовой базе становятся слишком сложными. В таких случаях вам будет необходимо взаимодействовать с командой дизайнеров для поиска оптимального решения. У вас уже настроена структура для разработки кастомных тем, поэтому теперь вы расширите ее с помощью корпоративного руководства по стилю. Начнем с доступности, так как ее следует закладывать в дизайны с самого начала.

Доступность

На доступность я обращаю внимание в первую очередь, поскольку о ней часто забывают, когда пытаются подготовить минимально жизнеспособный продукт (MVP, minimum viable product), и команда разработчиков быстро выпускает новые

функции. Доступность — не второстепенная часть дизайна приложения. Она, возможно, даже важнее, чем дизайн для мобильных устройств. Помимо предоставления равного доступа к информации для людей с ограниченными возможностями, доступность является юридическим требованием, согласно Закону США о запрете дискриминации людей с ограниченными возможностями (ADA, Americans with Disabilities Act) (https://oreil.ly/7piF5).

Хотя MUI предоставляет множество функций для обеспечения доступности, ваша ответственность — гарантировать использование таких элементов, как ARIA-метки, альтернативный текст для изображений, семантический HTML для скринридеров и комбинации клавиш для навигации по странице. Вот примеры фрагментов кода:

```
<!-- пример ARIA-метки -->
<button aria-label="Отмена" onClick={onCancel}>Отмена</button>

<!-- пример альтернативного текста для изображения -->
<img src="https://images.unsplash.com/photo-1497531551184-06b252e1bee1"
     alt="Разноцветный воздушный шар в небе с тремя людьми в корзине" />

<!-- пример семантического HTML -->
<h1>Добро пожаловать в тестовый магазин</h1>
```

При разработке форм убедитесь, что:

- инструкции сформулированы четко;
- сообщения информативны, включая сообщения об ошибках и успешных операциях.

Выявляйте области, где недостаточно цветового контраста, но помните, что за это должна отвечать команда дизайнеров. Статический контент должен легко находиться скринридерами, так как он часто содержит критически важную информацию.

Используйте семантические элементы вместо тегов

Если ваши компоненты состоят из множества `<div>`, замените их семантическими элементами (`<section>`, `<article>`, `<aside>` и др.) (https://oreil.ly/uFlOr). Разработчики часто используют ограниченный набор элементов, игнорируя остальные. Поскольку кастомные стили все равно потребуются, выбор семантических тегов не повлияет на визуальный рендеринг страницы.

Каждый HTML-элемент имеет значение для вспомогательных технологий, улучшая пользовательский опыт. Перед добавлением `<div>` проверьте, насколько легко передвигаться по приложению только с помощью скринридера и клавиатуры. Для тестирования используйте инструменты вроде Polypane (https://polypane.app/docs) или Responsively (https://responsively.app). Это не означает, что `<div>` плох, но всегда учитывайте, что, возможно, есть более подходящий элемент.

Создание доступной формы

Важно, чтобы принцип доступности был реализован в формах, поскольку они позволяют пользователям выполнять действия. Если кто-то не может понять форму, он не сможет выполнить критически важные задачи, такие как отправка платежей, обновление персональной информации или запрос услуг. Не забудьте также обеспечить доступность информационного контента, иначе пользователь с ограниченными возможностями может вообще не добраться до ваших форм.

В этом веб-приложении мы, с учетом доступности, создадим повторно используемую стилизованную форму, которая будет добавлена на страницу информации о пользователе для первой строки поиска. Предположим, вы обсудили это с коллегами и решили использовать React Hook Form (https://oreil.ly/H1lqR) в качестве пакета для всего веб-приложения. Установите его следующей командой:

```
npm install react-hook-form
```

После установки создайте новый компонент в директории src/elements с именем `SearchBar.tsx`. Он будет использоваться для всех строк поиска в веб-приложении. Вот код для строки поиска:

```
import { Input, InputAdornment, InputLabel } from '@mui/material';
import SearchIcon from '@mui/icons-material/Search';
import { useForm } from 'react-hook-form';
import styled from 'styled-components';

export type SearchBarProps = {
  name: string;
  onSubmitSearch: (searchText: string) => void;
};

const FullWidthForm = styled.form`
  width: 450px;
  @media (max-width: 500px) {
    width: 100%;
  }
`;

const SearchBar = (props: SearchBarProps) => {
  const { onSubmitSearch } = props;
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm();

  const searchFieldInputProps = {
    maxLength: 15,
    minLength: 3,
  };
```

```
  return (
    <FullWidthForm
      aria-label={`${props.name} форма поиска`}
      onSubmit={handleSubmit(onSubmitSearch)}
    >
      <InputLabel htmlFor="search">Поле поиска</InputLabel>
      <Input
        placeholder={`Поиск ${props.name}...`}
        type="search"
        fullWidth
        inputProps={searchFieldInputProps}
        startAdornment={
          <InputAdornment position="start">
            <SearchIcon />
          </InputAdornment>
        }
        {...register('search', { required: true, maxLength: 15, minLength: 3 })}
        aria-invalid={errors.search ? 'true' : 'false'}
      />
      {errors && errors.search && (
        <span>Текст поиска не соответствует требованиям</span>
      )}
    </FullWidthForm>
  );
};

export default SearchBar
```

Не забудьте создать файлы `index.tsx` и `SearchBar.test.tsx`! Больше не будем заострять на этом внимание – просто запомните, что шаблон для новых компонентов всегда включает эти файлы в дополнение к основному файлу компонента.

MUI и React Hook Form предоставляют формы, которые легко стилизуются, обладают доступностью «из коробки» и позволяют управлять полями через легковесный API. Вам потребуется изучить документацию MUI, чтобы полностью понять пропсы, передаваемые компоненту `<Input />` (https://oreil.ly/iQOBE), но в контексте доступности следует выделить несколько аспектов.

Все `ARIA-метки` (https://oreil.ly/YFXGI) помогают пользователям скринридеров отслеживать свое местоположение на странице. Поскольку не каждому элементу нужна `ARIA-метка`, используйте их только для интерактивных элементов. Большинство элементов должно быть интуитивно понятно благодаря видимому тексту, что обычно корректно интерпретируется вспомогательными устройствами. Кроме того, обратите внимание: в строке поиска отсутствует кнопка отправки. Это заложено в дизайне и улучшает доступность, поскольку стандартное поведение формы — отправка при нажатии клавиши Enter.

Пропс `type="search"` — это способ внедрения семантического HTML в компонент, так как он указывает, что элемент предназначен для выполнения поиска. Также

значимый элемент — пропс `inputProps` (https://oreil.ly/8XKA5). Это один из методов, с помощью которого MUI позволяет обрабатывать ошибки в формах.

Важный параметр доступности — отображение ошибок: оно помогает пользователю увидеть, что требуется исправить и как это сделать. Данное поле ввода требует указания значения для отправки и имеет ограничение по минимальному количеству символов. Если пользователь введет слишком маленький текст, рядом с полем появится понятное оповещение о проблеме. Глава 18 посвящена различным ошибкам во фронтенде и способам их обработки.

Проверка реализаций доступности

Когда потребуется проверить, насколько полно удовлетворены требования доступности, первым делом обращаются к встроенным в браузер инструментам разработчика (DevTools). В частности, Chrome предлагает эффективные инструменты для оценки доступности веб-приложения. Вы можете выполнить быстрый аудит, проверив группировку элементов, их метки и связанную функциональность, как показано на рис. 17.1. Также можно протестировать порядок перехода от элемента к элементу с помощью клавиши Tab при навигации только с клавиатуры.

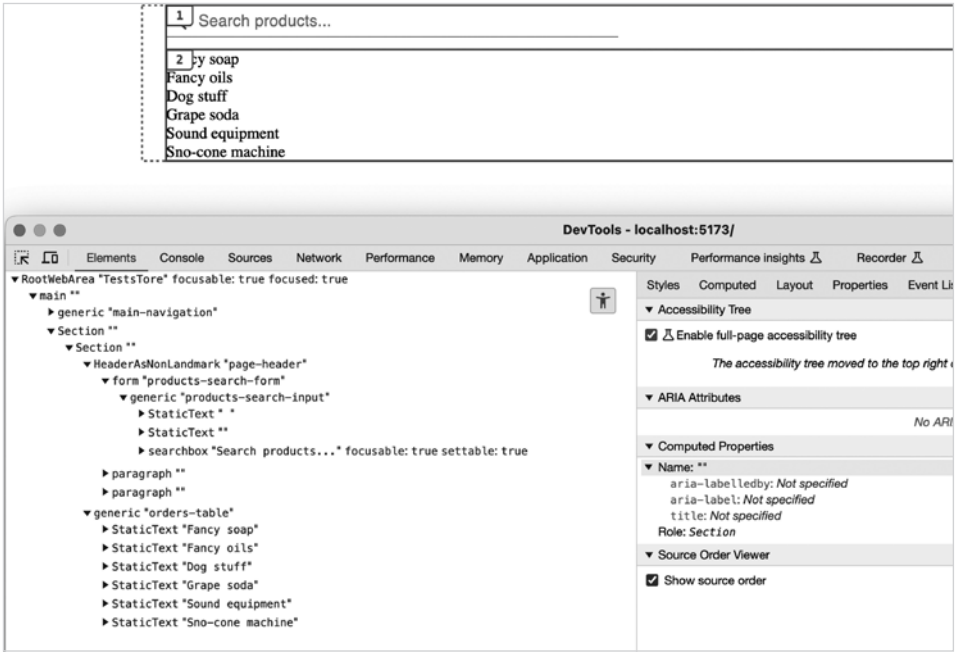

Рис. 17.1. Вкладка Accessibility в Chrome DevTools

Кроме того, вы можете провести аудит с помощью Lighthouse (https://oreil.ly/YNKNW) и получить больше информации о том, как улучшить доступность (рис. 17.2).

Lighthouse — это open-source инструмент, который проверяет производительность, доступность и другие метрики вашего веб-приложения. Если вы не уверены, что именно улучшать, он предоставит отличный список действий. Обсудите результаты вашей проверки с продуктовой командой и командой дизайнеров, так как могут возникнуть юридические аспекты, которые они курируют. Lighthouse не обязательно укажет точные компоненты для доработки, но даст общее представление о направлениях оптимизации.

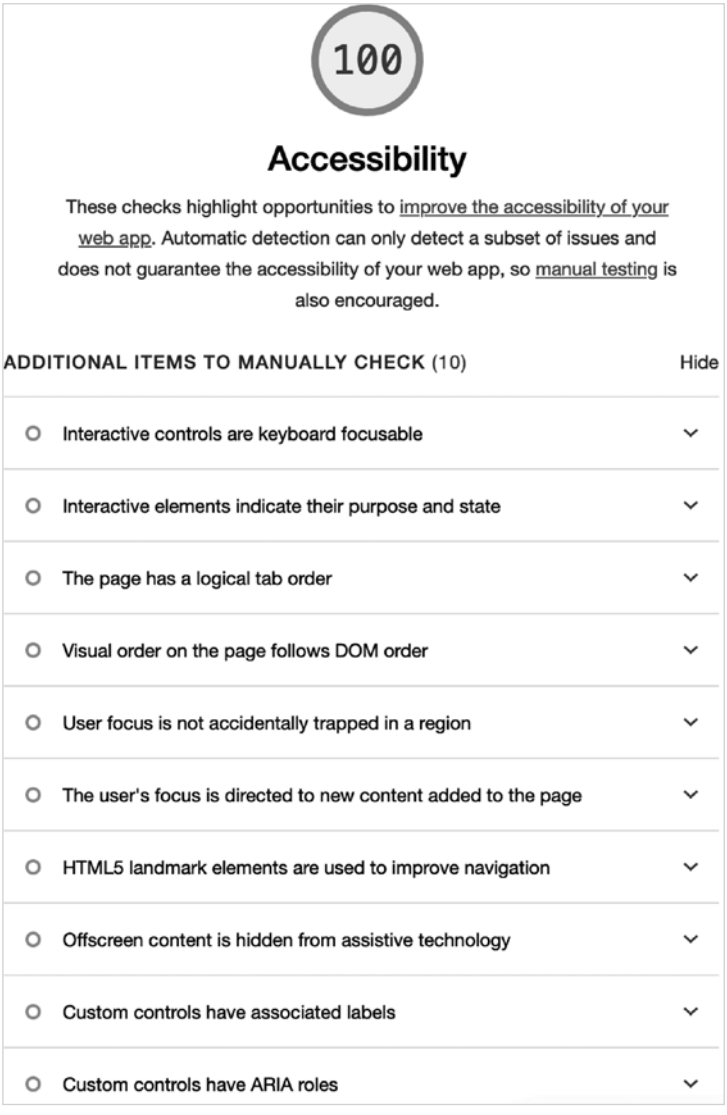

Рис. 17.2. Аудит доступности в Lighthouse

Еще один инструмент, который я использовала в корпоративных приложениях, — axe-core (https://oreil.ly/xI21p). Это небольшой пакет для добавления дополнительного слоя тестирования доступности. Вы можете установить его в свое веб-приложение и интегрировать в процесс тестирования во время разработки. Он поможет вам и команде находить часто пропускаемые ошибки в соблюдении правил доступности. Эта помощь особенно ценна при внедрении автоматизированного тестирования в пайплайн развертывания, который мы разберем в главе 24. Вот пример добавления в код:

```
axe.run(document, function (err, results) {
  if (err) throw err;
  console.log(results);
});
```

При запуске приложения этот код отобразит в консоли все проблемы с доступностью. Вы можете настроить axe для проверки только определенных правил WCAG (Web Content Accessibility Guidelines, Руководства по обеспечению доступности веб-контента, https://oreil.ly/Br9Qd) или для анализа конкретных компонентов приложения, если требуется сфокусироваться на сложных участках. Такой анализ полезен перед новыми релизами при подготовке к комплаенс-аудитам.

Дополнительные аспекты доступности

Большинство веб-приложений будут использоваться людьми, говорящими на языках, отличных от английского, поэтому важно учитывать и их. Отличный инструмент для такой локализации — i18n (https://oreil.ly/DPdXw). Вы можете создать файлы для каждого требуемого языка. Когда пользователь переключит приложение на другой язык, весь текст обновится мгновенно. При добавлении переводов необходимо учитывать дизайн: некоторым языкам требуется больше места для текста.

Унификация дизайнов

После того как вы решили вопрос с доступностью интерфейса приложения с учетом требований текущего дизайна и спецификаций, начинается череда непрерывных итераций дизайна. Сложность возникает, когда дизайн-материалы оказываются разбросаны по разным платформам — Figma, Miro или даже в виде скриншотов. Становится трудно отслеживать актуальные версии макетов. Постепенно появляются различия в общих компонентах, и внезапно обнаруживаются, например, четыре варианта дизайна для одного модального окна.

Хотя за управление и версионирование дизайнов обычно отвечает команда дизайнеров, именно вам предстоит реализовывать требования для новых функций. Заметив несоответствия, обязательно озвучьте их, независимо от того, насколько мелкими они кажутся: будь то изменение отступов у кнопок или разница цветовых оттенков в определенных разделах приложения. Подобные правки влияют на то, как вы и команда работаете над приложением.

Еще один аспект, который стоит обсудить в команде разработчиков, — это реализация стилей и их документирование. Возникнет множество мнений о том, как следует писать код. Одни разработчики считают инлайн-стили сложными для поддержки и выступают за использование стилизованных компонентов для каждого элемента. Другие не согласятся, предлагая определять стили через CSS-классы. Третьи предпочтут отдельные файлы стилей на уровне компонентов. Ни один из этих подходов не является ошибочным, и обычно применяется комбинация стратегий.

Главное, сохранять единообразие кодовой базы. При ревью пул-реквестов отмечайте любые отклонения от согласованных соглашений. Изменения неизбежны по мере роста приложения и ротации команды, но важно фиксировать в документации случаи отклонения от стандартов. Со временем код станет одним из вариантов контроля версий дизайнов, поэтому убедитесь, что все участники понимают суть изменений и могут объяснить, где они происходят.

Кастомизируемые темы

Вы уже реализовали кастомизируемые темы с помощью MUI в `theme.tsx` и будете поддерживать этот файл в актуальном состоянии при изменениях, затрагивающих компоненты MUI. Кнопки — характерный пример часто кастомизируемого компонента. Одни и те же кнопки используются во всем приложении и должны соответствовать брендингу компании. В текущих стилях несколько кнопок имеют фирменный зеленый цвет. Для их настройки обновите `theme.tsx` следующим образом:

```
…
const theme = createTheme({
…
  components: {
    MuiButton: {
      styleOverrides: {
        root: {
          color: '#4C4C4C',
          borderRadius: '4px',
          background: '#B4CD93',
        },
      },
    },
  },
})
```

Хотя MUI хорошо поддерживает кастомизацию тем, здесь могут возникать проблемы. Вы можете столкнуться с некоторыми конфликтами в MUI. Это характерно для любой выбранной библиотеки компонентов, однако неизбежно наступает момент, когда реализация дизайнов становится запутанной. Возникнут ситуации, когда элементы уже есть в ваших кастомизированных темах, но для конкретной страницы все равно потребуются инлайновые стили.

В результате в вашей теме и локальных компонентах образуется смешение правил CSS, из-за чего становится сложно понять, куда добавлять новые стили. Поэтому всякий раз, когда команда дизайнеров предлагает глобальные изменения для элементов, у которых уже есть кастомизированные стили, проговаривайте этот момент. Обычно такие правки требуют большего объема работы, чем предполагает команда дизайнеров, и кроме того, они могут неожиданно изменить другие части дизайна.

Важно помнить: не существует идеальной библиотеки компонентов или решения для стилизации. Есть другие варианты, например Tailwind CSS (https://oreil.ly/MpJ_z) или CSS Modules (https://oreil.ly/Cmr34), но реализация стилей — это скорее искусство, чем наука.

Каждый разработчик фронтенда имеет свое мнение, и команда дизайнеров также обычно предлагает свои идеи о наиболее эффективных решениях. Более того, вы можете столкнуться с ситуацией, когда вдруг обнаружите, что новый разработчик в команде не знает об имеющихся компонентах.

Инструменты вроде Storybook (https://oreil.ly/UsfYM), Ladle (https://ladle.dev) или React Cosmos (https://oreil.ly/g5r_h) помогают организовать документацию, упрощая поиск существующих элементов интерфейса. Это также позволяет команде дизайнеров оставаться в курсе уже реализованных компонентов. Кроме того, такие инструменты эффективны для прототипирования, особенно когда UI-блок имеет несколько вариантов. Их можно рассматривать как мост между разработчиками и дизайнерами для совместной работы над общими компонентами — что-то вроде брендбука для всей организации.

Адаптивный дизайн

Mobile-first дизайн (то есть ориентированный в первую очередь на мобильные устройства) — распространенная концепция, но ее не всегда применяют изначально. Иногда вы будете работать над приложениями, где сначала создают только десктопные макеты, а адаптивность добавляют позже. Поэтому, прежде чем приступать к разработке дизайнов исключительно для десктопов, уточните у продуктовой команды и команды дизайнеров, требуется ли также мобильная версия.

Есть несколько способов добавить адаптивность в проект постфактум, но это действительно потребует масштабного рефакторинга, независимо от выбранного подхода. Если вы используете библиотеку компонентов, как в нашем случае с MUI, то частичная адаптивность доступна как готовое решение. Остальное зависит от разработанной вами структуры. Один из подходов — использование *брейкпоинтов*, где стили или компоненты переключаются для определенных устройств, например планшетов и смартфонов. Вот пример реализации для компонента `Header`:

```
const StyledHeader = styled.header`
  display: flex;
  justify-content: space-between;
```

```
  @media (max-width: 500px) {
    flex-direction: column-reverse;
    width: 100%;
  }
```

Применение media queries (https://oreil.ly/iKwQk) — распространенный способ реализации адаптивности. Такой подход обычно работает хорошо, независимо от других стилевых решений, и его можно внедрить на любом этапе проекта. Кроме того, иногда целесообразно создавать для мобильной версии отдельные компоненты: если функциональность организована принципиально иначе, это упрощает чтение и поддержку кода. Соответственно, вы и команда можете разработать кастомные структуры макетов и таблицы стилей для страниц, где изменения очень значительны.

Создавайте компоненты с учетом адаптивности

Я работала над приложениями, где компания искренне считала, что мобильная или адаптивная версия не потребуется и что все пользователи будут использовать продукт исключительно на десктопе. Адаптивные версии таких приложений так и не были созданы. Хотя это редкость, подобное все еще происходит.

Чаще всего рано или поздно наступает такой момент, когда адаптивность приложения становится необходимой. Помните: это касается не только мобильных устройств и десктопов. Ваши пользователи могут работать с окном не на весь экран, использовать устройства с меньшими экранами, а их разрешение может отличаться. Также следует рассмотреть адаптивный дизайн с фиксированными макетами для конкретных типов устройств.

Если возможно, создавайте компоненты с учетом этого. Не обязательно добиваться идеала, если вы не следуете дизайн-макетам. Но если изначально учитывать адаптивность, это значительно упростит разработку в дальнейшем.

Учитывайте пользовательский опыт (UX) при адаптивном дизайне. Части интерфейса приложения могут перескакивать на страницу или полностью скрываться на маленьких экранах. Такие скачки могут сделать контент недоступным для пользователей, а продукт — непонятным или неудобным, особенно если пользователь перестает пользоваться мышью и вместо нее пользуется сенсорным экраном (тачскрином). Например, на тачскрине пользователю довольно сложно создать событие «при наведении» (hover).

Изображения — еще один аспект, требующий осторожности в адаптивном дизайне. Возможно, вы захотите растянуть или сжать изображение, чтобы оно вписалось

в определенную область в разумных пределах. Соотношение сторон должно сохраняться на всех размерах экранов для максимальной четкости изображения. С растровыми форматами (PNG, JPG) изображения могут стать зернистыми при растягивании. Для графики или иллюстраций используйте SVG (Scalable Vector Graphics), чтобы сохранить качество на любых устройствах. Пример реализации адаптивного изображения через `srcset` (https://oreil.ly/Pksox):

```
<img
  srcset="tulip-field-320w.jpg, tulip-field-480w.jpg 1.5x, tulip-field-640w.jpg 2x"
  src="tulip-field-640w.jpg"
  alt="Поле цветущих тюльпанов" />
```

Убедитесь, что проверяете поддержку CSS-свойств в браузерах через инструменты вроде CanIUse (http://caniuse.com). Не все версии браузеров поддерживают новейшие функции CSS, а некоторые из них не имеют обратной совместимости. Вам потребуется согласовать с продуктовой командой, какие браузеры каких версий следует поддерживать. В противном случае вы рискуете создать громоздкие CSS-стили лишь для поддержки Internet Explorer 11 у горстки пользователей.

Также рассмотрите возможность работы с веб-компонентами (Web Components) (https://oreil.ly/GPwzx). Это пользовательские элементы, создаваемые через JavaScript API. Они являются частью Shadow DOM, который изолирует стили и функциональность элементов от основного DOM, предотвращая конфликты. Это относительно новая технология, набирающая популярность во фреймворках JavaScript, поэтому она может упоминаться в обсуждениях об оптимизации адаптивного дизайна.

Заключение

В этой главе мы углубились в некоторые аспекты реализации дизайна нашего проекта. Среди всех элементов фронтенда этот, вероятно, вызывает больше всего споров. По-настоящему стандартного подхода не существует: каждая команда обладает уникальным опытом и представлениями о том, как сделать поддержку приложения удобной. По мере роста приложения, расширения организации и добавления новых участников в команду дизайнеров даже ключевые темы могут кардинально меняться. Главное здесь — стремиться к единообразию в пользовательском опыте (UX) и опыте разработчика (DX). Независимо от того, выберете вы готовые библиотеки компонентов, стилизацию через имена классов или их комбинацию, если это не нарушает UX, цель достигнута. А если команда понимает, откуда берутся стили и как они взаимосвязаны, DX остается контролируемым. Главное, поддерживайте постоянную коммуникацию между вашей командой, продуктовой командой и командой дизайнеров, чтобы минимизировать недопонимание.

ГЛАВА 18

Обработка ошибок во фронтенде

К этому моменту мы уже реализовали большую часть функциональности, типичной для фронтенд-приложений. Мы создали инфраструктуру, которая позволяет масштабировать приложение для поддержки новых функций, предлагаемых продуктовой командой. Теперь можно углубиться в детали фронтенда, повышающие удобство для пользователей.

В прошлых главах мы вскользь касались вопроса обработки ошибок и теперь наконец рассмотрим эту тему подробнее. Я намеренно откладывала ее на потом, чтобы вы могли сосредоточиться на оптимальных решениях для других частей фронтенда. Как бы то ни было, в реальных задачах требования должны включать обработку ошибок. Необходимо предусмотреть все потенциальные ошибки, с которыми может столкнуться пользователь. К реализации такой обработки следует подходить вдумчиво, чтобы не раскрыть конфиденциальную информацию ни пользователям, ни злоумышленникам, атакующим приложение, и чтобы обеспечить предсказуемость для пользовательского опыта.

В этой главе мы рассмотрим:

- компоненты ошибок;
- ошибки валидации пользователя;
- ошибки API.

Это стандартные ошибки, которые рано или поздно возникают, поэтому я расскажу о распространенных способах их обработки. Вам потребуется работать в связке с продуктовой командой и командой дизайнеров, чтобы определить и создать полезные состояния ошибок (error states), которые скорректируют движение пользователя в нужном направлении. По мере разработки приложения вместе с командой отмечайте любые пограничные случаи (edge cases), с которыми сталкиваетесь, — они могут навести на идеи о непредусмотренных сценариях.

Подходы к Error Boundary

Error boundary (компонент для перехвата ошибок) (https://oreil.ly/BYtEd) — это компоненты, отображающие резервный интерфейс (fallback UI), например сообщение об ошибке, при перехвате ошибок в дереве компонентов. Хотя этот

механизм уникален для React, но создать нечто подобное было бы удачным решением. Если у вас нет error boundary, то пользователи в продакшене обычно будут видеть пустой экран, а в процессе разработки вы будете наблюдать оверлей ошибок React с описанием проблемы. Это происходит при сбоях рендеринга компонентов или любых ошибках в среде выполнения, часто вызванных проблемами вызовов API, не предусмотренными действиями пользователя или сетевыми ошибками, которые могут возникнуть, если без определенного значения не получается сформировать запрос.

Можно было бы предположить, что сработает использование блоков `try/catch`, как в бэкенде, но они не перехватывают ошибки так, как нужно. Это происходит потому, что, в отличие от бэкенда, вызов функции делаете не вы, а React вместо вас, — особенность декларативного подхода. Такой способ обработки ошибок допустим для императивно реализованной функциональности, но для воспроизведения ошибок он небезопасен.

Разница между декларативным и императивным программированием

При обсуждении блоков `try/catch` в контексте обработки ошибок часто упоминаются декларативное и императивное программирование. *Декларативное программирование* — это когда вы указываете коду, что нужно сделать, но не указываете, как это реализовать. *Императивное программирование* — это когда вы прописываете конкретные шаги для достижения результата. Для наглядности рассмотрим обработку массивов:

```
const filteredOrders = orders.filter(order => order.totalAmount >= 100);
```

В императивном стиле решение выглядело бы так:

```
let filteredOrders = [];

for (let i = 0; i < orders.length; i++) {
  const order = orders[i];

  if (order.totalAmount >= 100) {
    filteredOrders.push(order);
  }
}
```

В React обычно применяют декларативный подход: с ростом опыта такой код читать легче, а разрабатывать — быстрее.

Чтобы обойти это ограничение с `try/catch`, вы можете применить несколько распространенных стратегий, создав error boundary. К ним относятся boundary на уровне:

- приложения;
- макета;
- компонента.

Независимо от уровня реализации error boundary, следует помнить, что таким образом не обрабатываются асинхронные ошибки, ошибки в обработчиках событий и ошибки, возникшие собственно в error boundary. В итоге вам потребуется комбинация этих стратегий для управления ошибками во всем приложении.

Error boundary *уровня приложения* располагается на вершине дерева компонентов проекта, охватывает все приложение и должен присутствовать в каждом проекте, чтобы обеспечить обработку ошибок с любого уровня дерева компонентов. Этот компонент также предотвращает аварийное завершение работы приложения. Он не детализирован и служит «ловушкой» для любых ошибок. Такой подход аналогичен использованию `try/catch` на верхнем уровне приложения с дополнительными блоками `try/catch` для конкретных ошибок.

Далее рассмотрим error boundary на *уровне макета*. Если у вас есть группа компонентов, например на странице пользователя в этом проекте, вы можете показывать пользователю более конкретизированные сообщения. Если подмножество компонентов использует какое-то общее состояние (shared state), добавьте error boundary на этом уровне. Учтите: при возникновении ошибки в одном компоненте это отобразится для всех компонентов макета.

Наконец, существуют error boundaries на *уровне компонента*. Это наиболее детализированный уровень, поскольку ошибка в отдельном компоненте не затрагивает остальные элементы страницы. При создании изолированных компонентов, например, с независимым состоянием или собственными вызовами API, реализуйте error boundary для них индивидуально.

Для реализации error boundary в React-приложении вы можете создать их с нуля (https://oreil.ly/NW692) или использовать готовые решения, например пакет `react-error-boundary` (https://oreil.ly/95Yyd). В этом проекте мы используем `react-error-boundary`, поскольку он легковесный и реализует удобную обертку, которая избавляет от необходимости сочетать классовые компоненты с функциональными. Установите пакет в репозиторий, затем в файл `App.tsx`, импортируйте его и внесите следующие изменения:

```
…
import { RouterProvider } from 'react-router-dom';
import { ErrorBoundary } from 'react-error-boundary';
import styled from 'styled-components'
…
const App = () => {
  // Здесь выполняется некоторая логика

  return (
```

```
    <ErrorBoundary
      FallbackComponent={ErrorFallback}
      onError={logError}
      onReset={() => window.location.reload()}
    >
      <QueryClientProvider client={queryClient}>
        …
      </QueryClientProvider>
    </ErrorBoundary>
  );
}
…
```

Этот подход оборачивает все приложение в error boundary. Таким образом, любые непредвиденные ошибки будут всплывать до верхнего уровня приложения и перехватываться. Вам потребуется создать компонент `Error Fallback` и функцию `logError`, используемые в error boundary. Дополнительно вы задействуете хук `useErrorBoundary` (https://oreil.ly/j_bs5) при возникновении ошибок в асинхронном коде или хуках управления жизненным циклом (например, `useEffect`).

Функция `onReset` обрабатывает повторные попытки после перехода в состояние ошибки. Ее поведение зависит от уровня размещения error boundary: обычно состояние компонента сбрасывается для повторного рендеринга. Если ошибка возникает на уровне приложения, вы можете использовать эту функцию для полного обновления страницы. Однако такой подход загружает данные из кэша, поэтому при проблемах с вызовом API потребуется реализовать error boundary на более низком уровне для перехвата соответствующего состояния.

Компоненты для обработки ошибок

Вам предстоит тесно сотрудничать с командой дизайнеров и продуктовой командой, чтобы определить визуальное представление ошибок. Среди методов обычно выделяют:

- модальные окна с кнопками повтора;
- полноэкранные сообщения, перенаправляющие пользователя;
- компоненты для тостовых уведомлений о временных ошибках;
- комбинацию этих методов.

В рамках этого проекта мы создадим специализированные компоненты, которые будут отображаться на странице вместо ожидаемой разметки.

Мы создадим компоненты для нескольких конкретных случаев, включая неопределенные значения, ошибки бэкенда и общие сбои. Рекомендуется начать с универсального компонента «Что-то пошло не так», чтобы приложение могло обработать любую неожиданность. Для совместимости с текущей обработкой ошибок в React Router v6 требуется рефакторинг `routes.tsx`.

Проверьте актуальность этого подхода в документации к React Router. Поскольку пакеты часто обновляются, к моменту чтения вами этой книги поведение могло измениться.

Текущее содержимое `routes.tsx`:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <UserInfo />,
  },
  {
    path: '/actions',
    element: <div>Actions go burr</div>,
  },
])
```

Обновите его, заменив `createBrowserRouter` на `useRoutes`:

```
import { useRoutes } from 'react-router-dom'
import UserInfo from './screens/UserInfo'

const Router = () => {
  const routes = useRoutes([
    {
      path: '/',
      element: <UserInfo />,
    },
    {
      path: '/actions',
      element: <div>Actions are here</div>,
    },
  ])

  return routes
}

export default Router
```

Затем обновите `App.tsx`, заменив:

```
import router from './routes'
import { RouterProvider } from 'react-router-dom'
…
<RouterProvider router={router} />
```

на:

```
import Router from './routes'
import { BrowserRouter } from 'react-router-dom'
…
<BrowserRouter>
  <Router />
</BrowserRouter>
```

Учитывайте, что некоторые пакеты содержат встроенные error boundaries, которые могут перехватывать исключения на неожиданном уровне. Например, React Router в текущей версии перехватывает ошибки на уровне `<RouteProvider>`. Если во время разработки экран ошибки отображается некорректно, проверьте участки кода, генерирующие ошибки, — возможно, их блокирует сторонний механизм.

Теперь, когда error boundary настроен, а код перестроен для его обработки, можно создать универсальный компонент `ErrorFallback`. В папке src/components создайте каталог ErrorFallbacks. В нем добавьте три файла: `index.tsx`, `ErrorFallback.tsx` и `ErrorFallback.test.tsx`. В `ErrorFallback.tsx` поместите следующий код:

```
import { Box, Typography, Button } from '@mui/material';
import { ErrorFallbackProps } from '../../types/errorTypes';

const ErrorFallback = ({ error, resetErrorBoundary }: ErrorFallbackProps) => {
  return (
    <Box
      margin="24px auto"
      role="alert"
      display="flex"
      flexDirection="column"
      gap="24px"
      textAlign="center"
    >
      <Typography variant="h2">Произошло что-то странное</Typography>
      <Typography variant="h3">Вот что произошло:</Typography>
      <Typography color="tomato" variant="body1">
        {error.message}
      </Typography>
      <Button variant="contained" onClick={resetErrorBoundary}>
        Обновите страницу для повторной попытки
      </Button>
    </Box>
  );
};

export default ErrorFallback
```

Не забудьте обновить файл `index.tsx` для экспорта компонента как модуля — это делается аналогично другим компонентам. Тесты следует писать параллельно с разработкой компонента, но мы вернемся к ним в главе 21, где разберем лучшие практики тестирования.

Теперь у вас есть компонент, который отобразится при перехвате ошибки на уровне компонента или макета. Он принимает объект ошибки и выводит сообщение на страницу — это помогает разработчикам понять причину сбоя. Результат будет аналогичен представленному на рис. 18.1 при возникновении ошибки времени выполнения.

Рис. 18.1. Универсальный компонент отображения ошибки с сообщением

Логирование ошибок

Второй аспект обработки ошибок — их логирование, чтобы сохранять записи о происходящем в приложении при отладке проблем в продакшене. Здесь помогут сторонние сервисы (например, Sentry, LogRocket или Datadog) или инструмент мониторинга от облачного провайдера, например Amazon CloudWatch. Для этого вы можете создать небольшую вспомогательную функцию. В папке src создайте новую директорию utils, а внутри нее — файл `helpers.tsx`. Добавьте в него следующую функцию:

```
import { ErrorFallbackProps } from '../types/errorTypes'

export const logError = (error: ErrorFallbackProps['error']) => {
  // настоятельно рекомендуется логировать во внешний инструмент
  // не выводите реальные ошибки в консоль
  console.log(error.message)
}
```

Поддерживайте единообразную структуру папок во фронтенде

Фронтенд достиг уровня, где выбор архитектуры и структуры папок стал сложнее и субъективнее, чем в бэкенде. Некоторые команды используют папку utils — ее также могут называть helpers или иначе. Вы можете выделить отдельную папку для типов или хранить их вместе с компонентом. Если команда не придерживается единообразия по мере роста приложения, поиск функциональности усложняется. Хотя в проекте используется конкретная структура папок (хорошая отправная точка для нового приложения), это не универсальное решение. Учитывайте: чем больше проектов и продуктов вы изучите, тем больше вариантов структур увидите и сформируете личные предпочтения. Не существует единственно верного подхода — важно, чтобы вся команда соблюдала согласованную структуру.

Логирование можно сделать настолько четким, насколько вы с командой посчитаете нужным. Чем качественнее ваши логи, тем быстрее вы сможете отследить первопричины ошибок. Также можно использовать инструмент логирования для получения метрик о наиболее частых ошибках. Это помогает выявить места, которые требуют рефакторинга или которыми неудобно или сложно пользоваться. Такие случаи могут возникать, если пользователи вынуждены вводить данные, опираясь на плохо сформулированные инструкции.

Ошибки валидации данных пользователя

Когда пользователи взаимодействуют с приложением, им требуется обратная связь о вводимых ими данных. Можно показывать ошибки прямо в интерфейсе, как только пользователь фокусируется на поле ввода. Или, как вариант, — откладывать показ до попытки отправить форму или до смены фокуса. Также допустимо выводить ошибки единым блоком. Скорее всего, вы будете комбинировать эти подходы, чтобы польза от информирования была максимальной.

Также необходимо учитывать используемую библиотеку компонентов. Поскольку в этом проекте применяется MUI, валидация полей ввода уже встроена — вы могли заметить это при создании формы поиска.

В MUI можно сделать поле обязательным и передать ему параметры валидации следующим образом:

```
const searchFieldInputProps = {
  maxLength: 15,
  minLength: 3,
  required: true,
}

<Input
  placeholder={`Search ${props.name}...`}
  type="search"
  fullWidth
  inputProps={searchFieldInputProps}
  startAdornment={
    <InputAdornment position="start">
      <SearchIcon />
    </InputAdornment>
  }
  {...register('search', { required: true, maxLength: 15, minLength: 3 })}
  aria-invalid={errors.search ? 'true' : 'false'}
/>
```

Теперь пользователь не может отправить запрос, введя менее 3 или более 15 символов в поле. Если пользователь попытается отправить форму, а ввод не соответствует требованиям валидации, он увидит конкретное сообщение об ошибке, как на рис. 18.2 и 18.3.

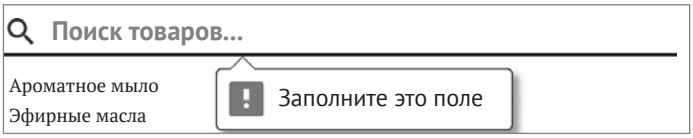

Рис. 18.2. Сообщение об ошибке обязательного ввода

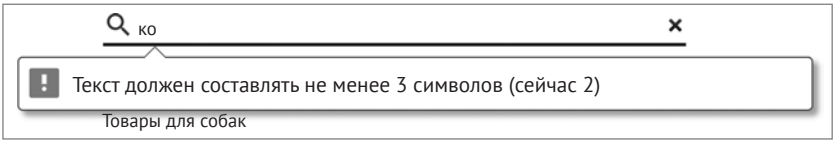

Рис. 18.3. Сообщение об ошибке ограничения длины строки

Эти сообщения поступают напрямую из стандартной валидации HTML в Chrome для полей ввода. Вы можете использовать другие форматы сообщений об ошибках (https://oreil.ly/KenbN), например выделение ввода в форме с кастомизированным сообщением. Способ обработки сообщений об ошибках значительно варьируется в зависимости от используемых компонентов для получения ввода от пользователя.

Другой широко используемый подход к валидации вводимых пользователем данных — проверка на уровне формы с помощью инструментов вроде React Hook Form. Вы можете добавить валидацию, которая проверит все поля при нажатии кнопки отправки. Реализация может выглядеть так:

```
<Input
  placeholder={`Поиск ${props.name}...`}
  type="search"
  fullWidth
  inputProps={searchFieldInputProps}
  startAdornment={
    <InputAdornment position="start">
      <SearchIcon />
    </InputAdornment>
  }
  {...register('search', {
    required: true,
    maxLength: 15,
    minLength: 3,
    pattern: /^[A-Za-z]+$/i,
  })}
  aria-invalid={errors.search ? 'true' : 'false'}
/>
```

Кроме того, вы можете использовать схему валидации с такими инструментами, как Zod (https://oreil.ly/HCRqv), joi (https://oreil.ly/dhDMH) или Yup (https://oreil.ly/3FHJz), совместно с React Hook Form, передав схему в `useForm` в качестве опции. Она будет

проверять введенные данные на соответствие схеме и отображать ошибки. Вот пример для формы поиска с использованием Yup:

```
import { useForm } from "react-hook-form";
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from "yup";

const schema = yup
  .object({
    searchText: yup.string().min(3).max(15).required(),
  })
  .required();

const SearchBar = (props: SearchBarProps) => {
  const { onSubmitSearch } = props;
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm({
    resolver: yupResolver(schema),
  });
```

Представление валидаций в виде функций или схем упрощает их повторное использование в различных компонентах и расширяет возможности кастомизации. В рамках пакета React Hook Form любой элемент `<input />`, зарегистрированный с валидациями, проверяется при активации функции `onSubmit`. Когда вы показываете сообщения пользователю на этом этапе сценария, ему проще понять необходимость внесения изменений перед повторным нажатием кнопки. Преимущество: пользователь не увидит ошибок до выполнения действия отправки.

Поскольку существуют разные взгляды на оптимальные способы отображения сообщений о валидации, крайне важно сотрудничать с командой дизайнеров — у них обычно более четкое понимание ожиданий пользователя от приложения. Ваша задача при этом — помочь им определить потенциальные ошибки бэкенда и их интерпретацию для пользователя.

Ошибки API

React Query, который вы используете для получения данных, уже содержит встроенную обработку ошибок, что упрощает работу с ними из бэкенда. В зависимости от проектирования API, коды состояния могут применяться для отображения информативных сообщений об ошибках как пользователям, так и разработчикам. Эти ошибки не обязательно связаны с действиями пользователя, но ему может потребоваться обновить страницу или предпринять другие действия для повторного выполнения запроса. Ваши компоненты ошибок должны направлять пользователя к решению проблемы, не раскрывая деталей происходящего в бэкенде.

Например, если API возвращает код состояния 422, это может указывать на проблему в форматировании пользовательского ввода для запроса. Есть вероятность, что фронтенд передает в бэкенд неверное имя параметра. Или, если сервер API недоступен, фронтенд получит код 500. В таких сценариях пользователь не может ничего исправить, но ему все равно нужно что-то показать. Здесь логирование окажется полезным для разработчиков, занимающихся отладкой. Вы и команда разработчиков можете даже привязать сообщения логов к каждому экрану ошибок API.

На рис. 18.4 показан пример получения ошибки 404 из запроса к бэкенду в инструментах браузера.

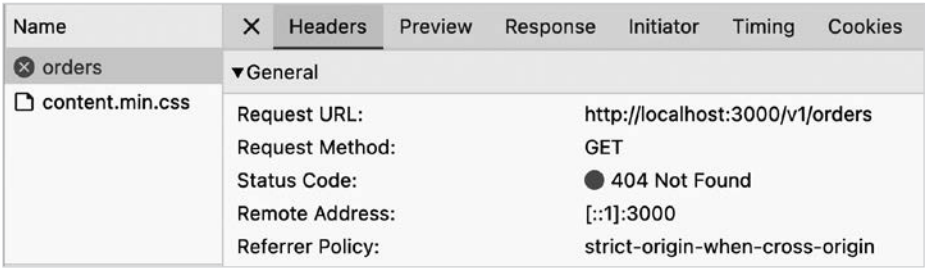

Рис. 18.4. Пример ошибки 404 от API-запроса

Вот как выглядит JSON-ответ для такого запроса:

```
{
    "statusCode": 404,
    "message": "Not found"
}
```

Эти экраны или компоненты будут похожи на уже созданный вами универсальный компонент ошибки; они станут его модифицированными версиями с другими формулировками и, возможно, кнопками действий. Основные отличия заключаются в том, что они отображаются только при заданных условиях и работают на уровне компонента или макета в приложении. Распространенные подходы к обработке содержимого включают:

- отображение сообщения об ошибке из бэкенда;
- показ напрямую из фронтенда кастомного сообщения, строго основанного на коде состояния, с деталями в логах.

Чтобы убедиться в корректной обработке ошибок API, используйте инструменты вроде httpstat.us (https://httpstat.us) или браузерное расширение tweak (https://oreil.ly/Vrxdi). Также можно имитировать ошибки сервера через Mock Service Worker (https://oreil.ly/Qv-ea) или добавить в запросы параметр, принудительно вызывающий определенную ошибку и возвращающий полезную нагрузку. Например:

```
GET /order/72?responseStatus=404
```

Это возвращает нас к дизайну API: сообщения об ошибках в ответах могут быть актуальны только для разработчиков при отладке, а пользователям или злоумышленникам не нужен доступ к точным деталям. Одна из стратегий — логировать сообщение об ошибке из бэкенда, но показывать пользователю иной текст. Будьте осторожны: злоумышленник или любопытный пользователь может увидеть ответ в инструментах браузера. Обычно во фронтенд отправляют отредактированную версию ошибки, чтобы избежать таких ситуаций и сосредоточиться на дружелюбных и полезных сообщениях для пользователей.

Заключение

В этой главе мы узнали о различных уровнях error boundary, сообщениях валидации пользовательских данных и о методах обработки ошибок API. Обработка ошибок критична для пользовательского опыта, отладки разработчиками и обеспечения безопасности приложения. Когда ваше приложение стабилизируется и начнутся обсуждения его вывода в продакшен, убедитесь, что у вас есть хотя бы базовый компонент для обработки общих ошибок. Этот аспект часто упускают из виду при фокусе на разработке функционала, если только продуктовая команда, команда дизайнеров или вы сами не поднимете вопрос.

Добавьте это в контрольный список работоспособности приложения. Возможно, вам потребуется изучить документацию React и документацию пакетов в приложении, чтобы полностью понять механизм обработки ошибок. Понимание источников ошибок, их распространения по приложению и мест отображения на странице повлияет на стиль написания кода всей командой. Такой момент очень удобен, чтобы организовать встречу с командой, показать и объяснить ей природу ошибок, а также обсудить проблемы, с которыми коллеги уже столкнулись. Обработка ошибок может быть сложной, но после ее внедрения вы и команда сможете устранить любые недоразумения.

ГЛАВА 19

Вопросы безопасности фронтенда

Безопасность — еще один аспект фронтенда, который необходимо учитывать в первую очередь при реализации новых функций. Безопасность крайне важна и настолько многогранна, что заслуживает отдельной книги, однако в этой главе мы рассмотрим достаточно, чтобы вы понимали, на что обращать внимание.

В главе 8 мы уже познакомились с некоторыми уязвимостями и методами обеспечения безопасности в бэкенде. Наиболее доступной частью продукта является фронтенд, и именно он может служить точкой входа для атак на сервер, поэтому теперь нам необходимо обратить внимание на такие аспекты, как уязвимости браузера и способы получения несанкционированного доступа, которыми могут воспользоваться злоумышленники, чтобы манипулировать работой приложения. Необходимо продумать, как вы будете хранить и передавать данные во фронтенде, поскольку все, что загружается в браузер, становится доступным пользователям, если они откроют инструменты разработчика. Вам и команде необходимо искать баланс между удобством пользователей и безопасностью.

В этой главе мы рассмотрим:

- дополнительные пункты из OWASP Top 10;
- распространенные векторы атак;
- методы взлома приложений;
- способы получения информации злоумышленниками напрямую из браузера;
- стратегии сокращения возможных атак.

Вам также необходимо будет ознакомиться с локальными и региональными законами о защите данных и нормативными требованиями, которые регулируют ваш продукт, такими как PCI DSS (https://oreil.ly/wAXSU), HIPAA (https://oreil.ly/ve3RM) и GDPR (https://gdpr.eu/checklist) (все они упоминались в главе 8). Эти нормативы могут быть строже общепринятых практик, учитывайте это.

Часто разработчикам не хватает базового понимания, как проводятся атаки. Вам не нужно углубляться в кибербезопасность или изучать множество инструментов, но представление о том, что искать, поможет разобраться в механизмах атак и способах их предотвращения.

Распространенные уязвимости

Причины многих уязвимостей фронтенда можно найти в *небезопасном проектировании* (https://oreil.ly/PVscX). Из главы 8 вы знаете, что бэкенд в большей степени отвечает за аутентификацию, авторизацию и множество настроек безопасности. Небезопасное проектирование часто возникает *случайно*. Вы и команда можете приложить все усилия, чтобы сделать приложение максимально защищенным, но жесткие сроки или нечеткие требования иногда приводят к появлению неожиданных уязвимостей. Рассмотрим некоторые часто упускаемые из виду аспекты.

Валидация бизнес-логики

Иногда уязвимости возникают из-за того, что недостаточно тщательно спроектирован сценарий взаимодействия пользователя (user flow). Даже *«идеальный путь»* — оптимальный сценарий, который вы и организация закладываете в приложение, — может содержать множество пограничных случаев, и существует опасность, что ими воспользуются злоумышленники.

Например, в вашем приложении таким уязвимым местом может быть процесс оформления заказа. Что произойдет, если пользователь попытается создать несколько заказов на один товар за несколько минут? Или отменит заказ сразу после его отправки со склада? Хотя такие сценарии выглядят экзотично, но их игнорирование подвергает компанию риску.

Представьте, что клиенты записываются на поддержку через календарь в приложении. Есть ли ограничение на количество сессий для одного пользователя? Если нет, кто-то может создать бота, который забронирует все сессии, что приведет к потерям времени и денег. Анализируя дизайн и спецификации, ищите подобные лазейки и продумывайте, как их эксплуатировать.

Формы отправки данных — частые векторы атак. Возможно, вам захочется сообщать пользователям о количестве оставшихся попыток ввода пароля или о том, в чем конкретно заключается ошибка ввода, но будьте осторожны с предоставляемой информацией. Злоумышленник может использовать ее для создания ботов. Защититься помогут скрытые лимиты попыток или дополнительные шаги вроде CAPTCHA.

Любые данные, передаваемые в бэкенд-запросах (например, query-параметры URL или тело запроса), могут стать вектором атаки (https://oreil.ly/EiiOe), если раскрывают параметры API-запросов (это не проблема, если нет чувствительных данных, а бэкенд проводит валидацию). Query-параметры полезны для создания совместно используемых ссылок, но нужно проследить, какая информация об API видна в URL.

Эти нюансы также поднимают вопрос о том, сколько данных передается во фронтенд. Старайтесь ограничивать объем запрашиваемых данных и возвращать только

то, что нужно пользователю, при этом стоит учитывать потенциально конфиденциальные (например, персональные) данные. В этот объем могут попадать данные, которые пользователь не видит сразу, но они могут понадобиться вскоре после запроса — для улучшения производительности. Даже если данные не загружаются на страницу, это не мешает людям проверять сетевые запросы в инструментах разработчика браузера. Работая над фронтендом и добавляя новые вызовы API, убедитесь, что они возвращают только необходимые данные, а конфиденциальная информация предоставляется только *после* подтверждения аутентификации и выполнения авторизации.

Управление сеансами

Управление сеансами (https://oreil.ly/A7qho) касается того, как вы обрабатываете и храните токены доступа во фронтенде, что также может быть вектором атаки (https://oreil.ly/ejtRk). Не все атаки происходят из-за того, что кто-то запустил ботов или набрал сложные команды в удаленном терминале. Атака может быть такой же простой, как проверка браузера на наличие авторизационных данных в URL или сохранение входа на старом устройстве. Просто вспомните, каким образом *вы* недавно заходили на свой любимый сайт или сервис.

При работе с персональными данными пользователей необходимо реализовать тайм-ауты сеансов. При использовании тайм-аутов неактивных сеансов (https://oreil.ly/BPo4U) пользователь автоматически выходит из системы, если в течение определенного времени не было новых запросов к API. Это помогает предотвратить перехват и использование идентификаторов сеансов (session ID) злоумышленниками. Проблема в том, что если атакующий *получит* действительный идентификатор сеанса, он может поддерживать сеанс активным и делать что угодно. Не надо хранить время сеанса в браузере, так как злоумышленник может легко изменить эти значения. Потребуется работа в бэкенде, чтобы убедиться, что по истечении времени идентификаторы становятся недействительными.

Также следует реализовать *абсолютные тайм-ауты сеансов* (https://oreil.ly/Tn230): независимо от того, где пользователь вошел в систему, через определенное время он принудительно выходит и должен снова аутентифицироваться. Это повышает безопасность, связанную с тайм-аутами неактивных сеансов, так как, даже если идентификатор сессии был перехвачен, он не будет действовать вечно. Это ограничивает период, в течение которого злоумышленник имеет доступ к учетной записи пользователя.

Не забудьте добавить кнопку «Выход» (Logout), чтобы пользователи могли завершать свои сеансы. Пользователи, заботящиеся о безопасности, захотят иметь возможность выходить самостоятельно. При наличии автоматических тайм-аутов это легко упустить из виду. Такую функциональность также можно добавить к событиям, например, закрытию браузера или страницы, если вы хотите реализовать напоминания о выходе.

Еще один вариант — тайм-аут обновления сеанса (https://oreil.ly/4BHRx). В этом случае идентификатор сеанса истекает, а новый генерируется в фоновом режиме и заменяется на фронтенде без каких-либо действий со стороны пользователя. Это позволяет пользователям оставаться в системе дольше, не давая злоумышленникам доступа к тому же идентификатору сеанса. (Я надеюсь, у вас именно так все и происходит, если вы залогинены в каких-либо приложениях (https://oreil.ly/4BHRx), в которых давно не проходили повторную аутентификацию.)

Еще одно решение, которое вам предстоит принять, — разрешать ли пользователям входить в приложение с разных устройств одновременно. Разрешение входа с нескольких устройств потенциально позволяет злоумышленнику войти вместе с пользователем и оставаться незамеченным какое-то время. Если вы разрешаете это, настройте мониторинг и оповещения для выявления необычной активности между сеансами. С другой стороны, если одновременно разрешен только один вход, потребуется дополнительная работа на бэкенде, чтобы гарантировать аннулирование идентификаторов сеансов и реализовать механизм выбора устройства, которое должно остаться залогиненным.

Обслуживание версий пакетов

Устаревание версий пакетов — распространенная уязвимость (https://oreil.ly/EwZ87) как на фронтенде, так и на бэкенде. Если для пакета выпускались обновления безопасности (патчи), каждая уязвимость в устаревшей, необновленной версии хорошо документирована и публично доступна в различных списках CWE (Common Weakness Enumeration) (https://oreil.ly/k9vgY). Некоторые злоумышленники просто берут OWASP Top 10 и составляют список атак на основе этих данных.

Обновление версий пакетов часто откладывают. Причины разные: где-то это нежелание вникать в критические изменения, где-то — требование менять пакеты только в полном объеме. Рекомендую создавать тикет хотя бы раз в месяц для обновления всех пакетов до последней стабильной версии. В идеале вам и команде стоит обновлять пакеты параллельно с разработкой новых функций или устранением технического долга.

Можно использовать инструменты вроде Dependabot (https://oreil.ly/OyWhx), которые помечают уязвимые пакеты прямо в репозитории. Другие сканеры кода, такие как npm-audit (https://oreil.ly/Lzce_), Snyk (https://snyk.io) или retire.js (https://oreil.ly/8XXHN), показывают, какие пакеты устарели. Не все пакеты требуют постоянного сопровождения. Есть множество пакетов, которые отлично выполняют свою задачу, но не обновлялись годами. С опытом вы научитесь находить баланс при выборе пакетов.

Некоторые из упомянутых инструментов отлично подходят для интеграции в ваш CI/CD-пайплайн. Подробнее об этом мы поговорим в главе 27.

На рис. 19.1 показан пример отчета, который вы увидите при обнаружении уязвимостей после запуска `npm audit` в своих проектах.

```
# npm audit report

axios  1.3.2 - 1.7.3
Severity: high
Server-Side Request Forgery in axios - https://github.com/advisories/GHSA-8hc4-vh64-cxmj
fix available via `npm audit fix`
node_modules/axios

body-parser  <1.20.3
Severity: high
body-parser vulnerable to denial of service when url encoding is enabled - https://github.com/advisories/GHSA-qwcr-r2fm-qrc7
fix available via `npm audit fix`
node_modules/body-parser
  express  <=4.19.2 || 5.0.0-alpha.1 - 5.0.0-beta.3
  Depends on vulnerable versions of body-parser
  Depends on vulnerable versions of path-to-regexp
  Depends on vulnerable versions of send
  Depends on vulnerable versions of serve-static
  node_modules/express

braces  <3.0.3
Severity: high
Uncontrolled resource consumption in braces - https://github.com/advisories/GHSA-grv7-fg5c-xmjg
fix available via `npm audit fix`
node_modules/braces


follow-redirects  <=1.15.5
Severity: moderate
follow-redirects' Proxy-Authorization header kept across hosts - https://github.com/advisories/GHSA-cxjh-pqwp-8mfp
fix available via `npm audit fix`
node_modules/follow-redirects

micromatch  <4.0.8
Severity: moderate
Regular Expression Denial of Service (ReDoS) in micromatch - https://github.com/advisories/GHSA-952p-6rrq-rcjv
fix available via `npm audit fix`
node_modules/micromatch

path-to-regexp  <=0.1.9 || 4.0.0 - 6.2.2
Severity: high
path-to-regexp outputs backtracking regular expressions - https://github.com/advisories/GHSA-9wv6-86v2-598j
path-to-regexp outputs backtracking regular expressions - https://github.com/advisories/GHSA-9wv6-86v2-598j
fix available via `npm audit fix`
node_modules/msw/node_modules/path-to-regexp
```

Рис. 19.1. Фрагмент отчета об уязвимостях

Крайне важно проверять выбранные пакеты с самого начала. Поскольку это код, который вы не поддерживаете напрямую, вы должны быть уверены, что это делают другие. Если окажется, что один из используемых вами пакетов больше не поддерживается, это может привести к появлению уязвимости в вашем приложении. Ищите замену или рассмотрите возможность реализации функциональности собственными силами.

Проверка входных данных

Проверка входных данных будет немного отличаться от методов, которые мы рассматривали в главе 18, потому что теперь нас волнует не то, что видит пользователь. Нам важно, как код обрабатывает полученные данные. Вам необходимо программно проверять и очищать (https://oreil.ly/Np82L) данные, полученные от пользователей. Убедиться, что пользователь ввел корректный тип данных, можно с помощью валидационных схем и функций, реализованных в полях ввода и формах.

Однако вам также нужно проверять, не пытается ли пользователь навязать определенный формат значений, чтобы заставить приложение выполнить несанкционированное действие. Это может включать атаки типа SQL-инъекций (https://oreil.ly/20NJU) и межсайтового скриптинга (XSS) (https://oreil.ly/x5UHO). Подобные атаки (https://oreil.ly/VSu9E) распространены, поскольку подмена значений — это простой способ эксплуатации уязвимостей.

В вашем бэкенде уже реализованы валидация входных данных (https://oreil.ly/20NJU, https://oreil.ly/x5UHO, https://oreil.ly/VSu9E) и санитизация, но полезно добавить валидацию и на фронтенде, чтобы обеспечить мгновенную обратную связь для пользователя без нагрузки на сервер.

Любой пользователь может ввести что угодно в форму и нажать «Отправить» просто из любопытства. Валидация и санитизация на фронтенде улучшают пользовательский опыт (UX) и могут замедлить некоторые типовые атаки. Однако всегда помните, что ограничения формы легко обойти.

Вы можете выполнить валидацию формы с помощью встроенных атрибутов полей, таких как `required`, `type`, `min` и `max`. Это один из методов, которые вы уже использовали в проекте. Вот пример формы с такой валидацией, которую можно добавить в проект для последующей доработки в `src/elements/Forms/OrderForm.tsx`:

```
const OrderForm = () => (
  <form
    style={{
      display: 'flex',
      flexDirection: 'column',
      gap: '12px',
      alignItems: 'stretch',
      padding: '32px',
      width: '350px',
    }}
  >
    <div>
      <label>First Name: </label>
      <input type="text" required />
    </div>
    <div>
      <label>Last Name: </label>
      <input type="text" required />
    </div>
    <div>
      <label>Quantity: </label>
      <input type="number" min={0} max={12} />
    </div>
    <div>
      <label>Email: </label>
      <input type="email" />
    </div>
    <div>
```

```
      <label>Password: </label>
      <input type="password" />
    </div>
    <div>
      <label>Best Contact Time: </label>
      <input type="datetime-local" />
    </div>
    <button onSubmit={() => {}}>Submit</button>
  </form>
)
```

Это потребует от пользователя ввода определенных значений в различные поля формы перед отправкой. Также можно использовать программный подход к валидации, применяя пакеты для обработки форм, такие как React Hook Form. Вот несколько примеров использования этого пакета с компонентом `SearchBar`:

```
const schema = yup
  .object({
    searchText: yup
      .string()
      .min(3)
      .max(15)
      .required(),
  })
  .required();

{
  ...register('search', {
    required: true,
    maxLength: 15,
    minLength: 3,
    pattern: /^[A-Za-z]+$/i,
  })
};
…
```

Можно использовать схему валидации на всех полях ввода для большей гибкости и настройки или размещать правила валидации непосредственно в пропсах поля ввода.

Прочие принципы

В мире кибербезопасности существует выражение: «безопасность через неясность» (https://oreil.ly/uAd8Q). Оно означает, что только сотрудники, работающие с определенными частями системы, обладают знаниями о них. Например, команда ИБ обычно разбирается в ролях и реализации авторизации лучше, чем вы. Предоставление доступа к информации о механизмах только тем, кому это необходимо, помогает снизить угрозы эксплуатации уязвимостей. Еще один пример безопасности через неясность — использование корпоративного инструмента для хранения

паролей, где доступ к определенным паролям есть только у ограниченного круга сотрудников. Технически все доступно, но безопасность выше, поскольку пароли получают только те, кому они действительно нужны.

Я уже упоминала это в разделе об управлении сеансами, но повторю еще раз: *любые данные*, отправляемые с сервера клиенту и раскрывающие детали инфраструктуры, могут стать подсказками для злоумышленников. Даже такие мелочи, как заголовки `X-Powered-By` или URL, указывающие на хостинг-провайдера (например, стандартные эндпоинты AWS вместо кастомных доменов), могут выдать достаточно информации для атаки. Всегда следите за тем, какие данные попадают во фронтенд, — их легко обнаружить.

Как проверить свое приложение

Изучение основ этичного хакинга для анализа собственных приложений поможет взглянуть на безопасность с другой стороны. Вы начнете замечать, как уязвимости открывают пути для атак. Один из лучших ресурсов — PortSwigger Web Security Academy (https://oreil.ly/VdUzs). На этом сайте собраны подробные статьи, примеры и практические задания по распространенным атакам. Попробуйте, например, практикум по обходу двухфакторной аутентификации (2FA) (https://oreil.ly/Vc41o) или задание на низкоуровневые логические уязвимости (https://oreil.ly/C7f9d).

Выполнение практических заданий — лишь один из способов начать понимать, как мыслят злоумышленники и на что они обращают внимание при поиске уязвимостей. Это не то, что можно реализовать простым кодом, — здесь требуются развитые навыки наблюдения, умение находить закономерности и большая доля креативности. Именно в этом заключается удовольствие от этичного хакинга: он заставляет ваш мозг работать в новых направлениях. Погружение в эту тему поможет вам расти профессионально.

Если вы решите глубже изучить методы атак на разрабатываемые приложения, обратите внимание на Kali Linux (https://oreil.ly/Ie5c1). Эта ОС включает практически все необходимые инструменты для тестирования различных систем. Ее можно запускать на виртуальной машине, чтобы экспериментировать без установки отдельной операционной системы. Лучший способ научиться защищать свое приложение — понять, какие инструменты и методы используют злоумышленники для поиска и эксплуатации потенциальных уязвимостей. Чтобы узнать, какие уязвимости интересуют организации, изучите программы bug bounty на таких платформах, как HackerOne (https://oreil.ly/BJbAF) и Bugcrowd (https://oreil.ly/HRka-), а также на корпоративных сайтах, например Microsoft (https://oreil.ly/2VDU5) и Apple (https://oreil.ly/4KGS4).

Помните о необходимости соблюдать закон и учитывать юридические ограничения, если решите протестировать реальное приложение. Неосторожные действия могут привести к тому, что вы получите несанкционированный доступ к базе данных организации, а вашу атаку отследят по IP-адресу.

Заключение

В этой главе вы узнали больше о безопасности во фронтенде и о том, как злоумышленники получают доступ к данным пользователей и повышают уровень своих привилегий в системах. Они методично изучают фронтенд в поисках уязвимостей. Не думайте, что ваше приложение слишком малозначительно, чтобы заинтересовать злоумышленников, или настолько защищено, что неуязвимо. Даже если внутренняя логика системы неизвестна внешним наблюдателям, они могут восстановить ее по деталям. Работайте вместе с командой ИБ, чтобы обеспечить защиту фронтенда — первой линии обороны. Поскольку вы и ваша команда отвечаете за фулстек, важно понимать, как системы взаимодействуют и защищают пользовательские данные.

ГЛАВА 20

Производительность фронтенда

Теперь ваш фронтенд работает в хорошем окружении и вы, вероятно, уже выпустили как минимум бета-версию продукта для пользователей. В дальнейшем вы будете добавлять новые функции, заниматься поддержкой и искать способы улучшить приложение с точки зрения опыта как разработчиков, так и пользователей. Один из способов достичь обеих целей — повысить производительность. Чем быстрее и бесперебойнее работает приложение, тем лучше становится пользовательский опыт и тем проще разработчикам вносить изменения при локальной разработке.

Пользователей раздражают приложения, которые долго грузятся и интерфейс которых по мере загрузки данных «скачет». Команда, в свою очередь, тоже нервничает, если процесс обновления сопряжен со сложностями. Поэтому, как только работа приложения стабилизируется и основная архитектура будет готова, можно сосредоточиться на упомянутых проблемах. На этом этапе появляется отличная возможность оптимизировать код с учетом паттернов, обнаруженных в логах, а также отзывов команды разработчиков и других заинтересованных сторон.

В этой главе мы рассмотрим, как:

- устранять основные причины проблем с производительностью;
- измерять и улучшать метрики производительности;
- пользоваться инструментами для оптимизации работы приложения;
- поддерживать чистоту кода в целях повышения производительности.

Вопросы производительности обычно поднимаются в обсуждениях, когда приложение уже некоторое время работает в продакшене. Например, кто-то из разработчиков может пожаловаться, что разработка идет медленно из-за долгой загрузки изменений кода. Об этом стоит задуматься с самого начала, поскольку это напрямую влияет на поддержку и развитие приложения. Хотя многие проблемы с производительностью можно решить даже на поздних этапах проекта, чем раньше вы ими займетесь, тем лучше.

Базовые метрики

Прежде чем приступать к оптимизации, нужно понять, на какие метрики ориентироваться. Если вы не уверены, с чего начать, хорошей отправной точкой станут Core

Web Vitals (https://oreil.ly/y3YK4). Затем можно изучить специфические метрики, наиболее актуальные для вашего приложения. Влияет ли размер бандла на время загрузки? Как скачут показатели отклика на медленных сетях? Как обстоят дела с *Largest Contentful Paint* (LCP) (https://oreil.ly/zuIuG)? Можно сосредоточиться на нескольких метриках и постепенно вносить изменения в приложение. Некоторые из ключевых метрик для начала работы — это LCP (время отображения самого крупного объекта, Largest Contentful Paint), FID (задержка первого ввода, First Input Delay) и CLS (кумулятивный сдвиг макета, Cumulative Layout Shift).

LCP измеряет, сколько времени требуется для отображения самого крупного изображения или блока контента относительно момента перехода пользователя на страницу. Целевое значение — достичь LCP в пределах 2,5 секунды с момента начала загрузки страницы. Рекомендуется удерживать этот показатель хотя бы ниже 4 секунд. Более длительное время приводит к плохому пользовательскому опыту и требует улучшений со стороны команды.

FID измеряет, сколько времени браузеру требуется для обработки событий, вызванных действиями пользователя, другими словами, как быстро страница реагирует на клик или ввод данных. Оптимальное значение для этой метрики — 100 миллисекунд или меньше; все, что превышает 300 миллисекунд, требует доработки. Никому не нравится кликать на элемент и не получать никакой реакции в ответ.

CLS крайне важен при разработке макетов. Эта метрика оценивает, насколько «прыгает» страница во время загрузки. Необходимо минимизировать сдвиги макета при взаимодействии пользователя со страницей. Здесь полезны такие компоненты, как скелетоны загрузки, поскольку они сохраняют структуру макета во время подгрузки контента. Хороший показатель — менее 0,1; значения выше 0,25 требуют оптимизации. Следите за этой метрикой особенно тщательно, так как на нее разработчики могут влиять напрямую.

Инструменты тестирования производительности также помогут проанализировать другие метрики и выявить сильные стороны вашего приложения. Если вам интересно проверить часто посещаемые вами сайты, воспользуйтесь WebPageTest (https://webpagetest.org) или Google PageSpeed Insights (https://oreil.ly/WlGbs), введите URL, и они сгенерируют отчет о производительности.

Далее в этом разделе вы познакомитесь с двумя инструментами: Lighthouse (https://oreil.ly/IG2He) и sitespeed.io (https://oreil.ly/WnBxy).

Lighthouse

Вы, возможно, уже использовали Lighthouse в Chrome DevTools во время локальной разработки. Этот аудит выполняется быстро, поэтому проводите его периодически. Так вы сможете предоставлять команде разработчиков небольшие отчеты о потенциальных проблемах, на которые стоит обратить внимание при принятии решений. Это один из вариантов наставничества через реальные и практически

применимые примеры. Результаты также можно показать продуктологам и другим заинтересованным сторонам, чтобы они понимали, для каких целей вы запрашиваете время на работу с техническим долгом.

Хотя проверка результатов Lighthouse локально допустима, учитывайте, что показатели в среде разработке могут значительно отличаться от таковых в среде продакшена из-за ассетов без сжатия, «разбухшего» кода (https://oreil.ly/YBoJ3), неоптимальных конфигураций и прочего. Один из вариантов решить эту проблему — сделать продакшен-сборку локально и запустить Lighthouse для нее.

Если запустить текущий проект локально и открыть Chrome DevTools, можно выполнить тест производительности Lighthouse. Результаты будут похожи на рис. 20.1 и 20.2.

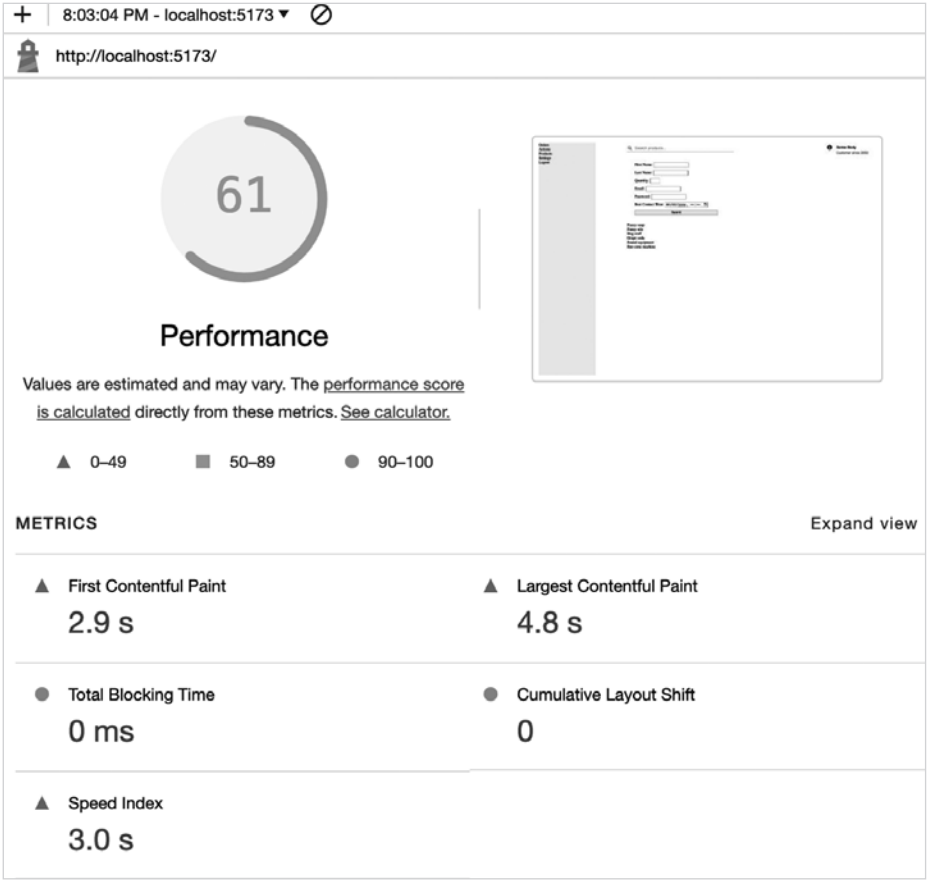

Рис. 20.1. Обзор отчета Lighthouse

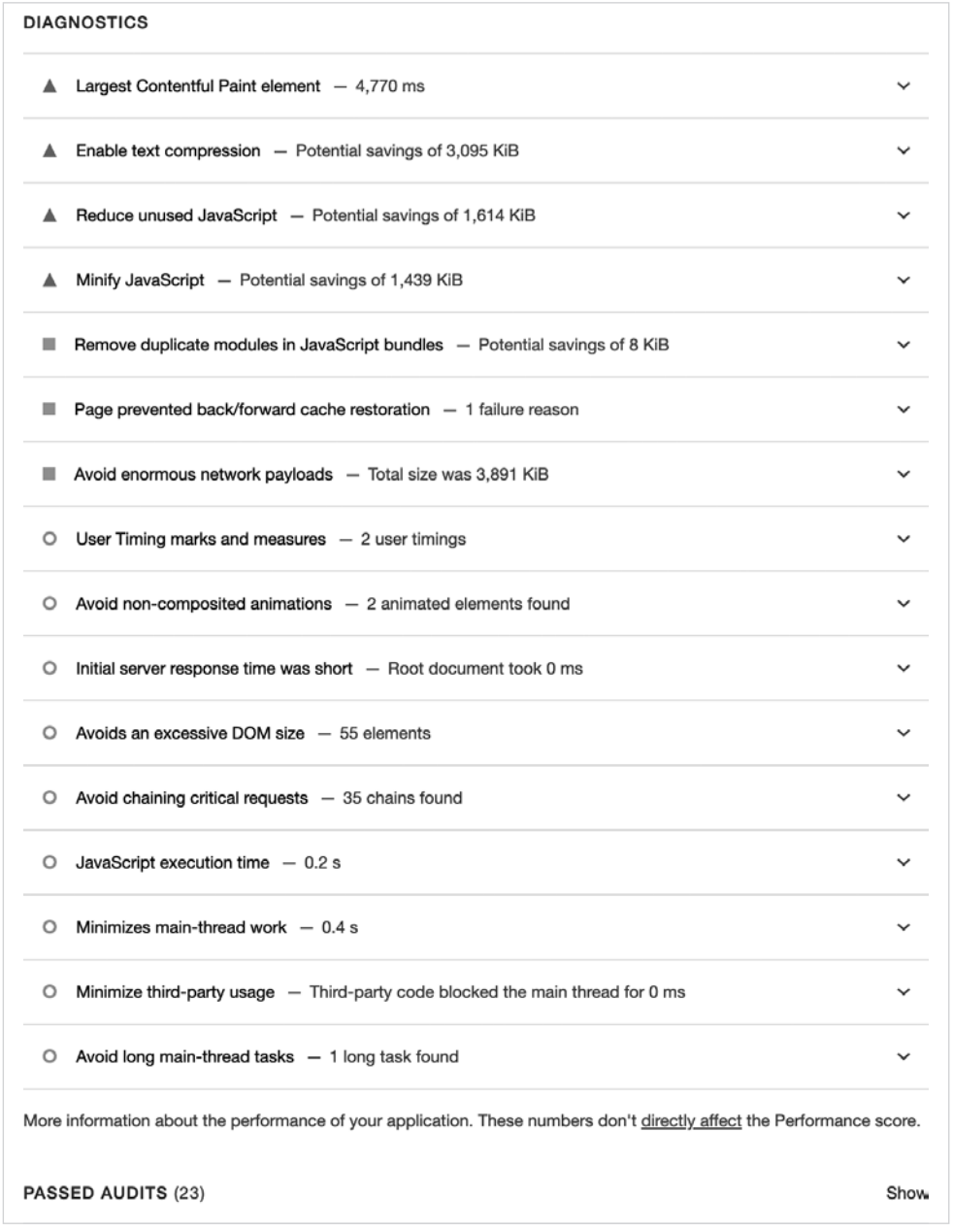

Рис. 20.2. Детали отчета Lighthouse

Теперь у вас есть список метрик для улучшения, а также рекомендации из отчета. Для автоматизации можно использовать npm-пакет Lighthouse (https://oreil.ly/

AN-nL), чтобы интегрировать отчеты о производительности в CI/CD-тестирование. Доступны различные конфигурации для генерации отчетов в форматах JSON или HTML. Для тестирования установите и запустите Lighthouse локально, сохранив результаты в JSON:

```
npm install -g lighthouse
lighthouse http://localhost:5173/ --output-path=./report.json --output json
```

Полный JSON-отчет доступен в репозитории проекта на GitHub (https://oreil.ly/C8omi). Ниже его фрагмент:

```
"first-contentful-paint": {
      "id": "first-contentful-paint",
      "title": "First Contentful Paint",
      "description": " First Contentful Paint (FCP) — это момент времени, когда на
странице впервые отрисовывается текст или изображение. [Подробнее о метрике First
Contentful Paint](https://developer.chrome.com/docs/lighthouse/performance/first-
contentful-paint/).",
      "score": 0,
      "scoreDisplayMode": "numeric",
      "numericValue": 14375.424124999987,
      "numericUnit": "millisecond",
      "displayValue": "14.4 s",
      "scoringOptions": {
        "p10": 1800,
        "median": 3000
      }
    },
    "largest-contentful-paint": {
      "id": "largest-contentful-paint",
      "title": "Largest Contentful Paint",
      "description": " Largest Contentful Paint (LCP) — это момент времени, когда
на странице отрисовывается самый крупный текстовый или графический элемент.
[Подробнее о метрике Largest Contentful Paint] (https://developer.chrome.com/docs/
lighthouse/performance/lighthouse-largest-contentful-paint/)",
      "score": 0,
      "scoreDisplayMode": "numeric",
      "numericValue": 25557.863749999982,
      "numericUnit": "millisecond",
      "displayValue": "25.6 s",
      "scoringOptions": {
        "p10": 2500,
        "median": 4000
      }
    },
```

Это те же результаты, которые вы увидите на HTML-странице, но теперь вы можете программно работать с ними. Возможно, есть определенные метрики, которые требуют особого внимания, поэтому вы можете настроить систему так, чтобы она генерировала ошибку, если значение метрики не превышает заданный порог. Это могут быть ключевые веб-метрики (Core Web Vitals), такие как LCP (Largest

Contentful Paint), FID (First Input Delay) и CLS (Cumulative Layout Shift), поскольку они существенно влияют на пользовательский опыт.

Sitespeed.io

Давайте рассмотрим open-source-инструмент, который дает больше гибкости при настройке и тестировании производительности.

Sitespeed.io — это действительно универсальное решение, поскольку с его помощью можно:

- создавать дашборды (https://oreil.ly/xCa6-) для мониторинга производительности приложения в динамике;
- получать отчеты по отдельным страницам;
- тестировать в разных браузерах.

Инструмент предлагает множество настраиваемых конфигураций для проверки практически любых метрик производительности, а также тестирования доступности. Sitespeed.io можно даже запускать в изолированном Docker-контейнере.

При необходимости можно настроить отправку оповещений команде в Slack-канал при превышении пороговых значений. Кроме того, инструмент поддерживает запись видео во время тестов производительности и легко интегрируется в CI/CD-пайплайны.

Рекомендую изучить документацию sitespeed.io (https://oreil.ly/wtM7-), чтобы узнать все возможности этого пакета. Ниже приведен пример теста:

```
npm i -g sitespeed.io
sitespeed.io http://localhost:5173 \  --browser safari \  -n 2 \  --summary-detail

[2024-02-17 19:32:04] INFO: Versions OS: darwin 23.1.0 nodejs: v21.2.0
                 sitespeed.io: 33.0.0 browsertime: 21.2.1 coach: 8.0.2
[2024-02-17 19:32:05] INFO: Running tests using Safari - 2 iteration(s)
[2024-02-17 19:32:06] INFO: Testing url http://localhost:5173 iteration 1
[2024-02-17 19:32:12] INFO: Take after page complete check screenshot
[2024-02-17 19:32:15] INFO: http://localhost:5173 TTFB: 3ms DOMContentLoaded: 196ms
FCP: 243ms Load: 197ms
[2024-02-17 19:32:16] INFO: Testing url http://localhost:5173 iteration 2
[2024-02-17 19:32:23] INFO: Take after page complete check screenshot
[2024-02-17 19:32:26] INFO: http://localhost:5173 TTFB: 3ms DOMContentLoaded: 201ms
FCP: 248ms Load: 201ms
[2024-02-17 19:32:26] INFO: http://localhost:5173 TTFB: 3ms (σ0.00ms 0%), FCP:
246ms (σ3.00ms 1.0%), DOMContentLoaded: 199ms (σ3.00ms 1.3%), CPUBenchmark: 61.5ms
(σ13.50ms 22.0%), Load: 199ms (σ2.00ms 1.0%) (2 runs)
[2024-02-17 19:32:27] INFO: HTML stored in /Repos/dashboard-ui/sitespeed-result/
localhost/2024-02-17-19-32-04
1 page analysed for http://localhost:5173 (2 runs, Safari/desktop/[Object object])
   Score / Metric             Median
   -------------              ------
```

```
√  First Contentful Paint     246 ms
   Page Load Time             199 ms
   TTFB                       3 ms
!  Coach Overall Score        84
✗  Coach Best Practice Score  68
!  Coach Privacy Score        80
√  Coach Performance Score    99
```

Результаты представлены на рис. 20.3.

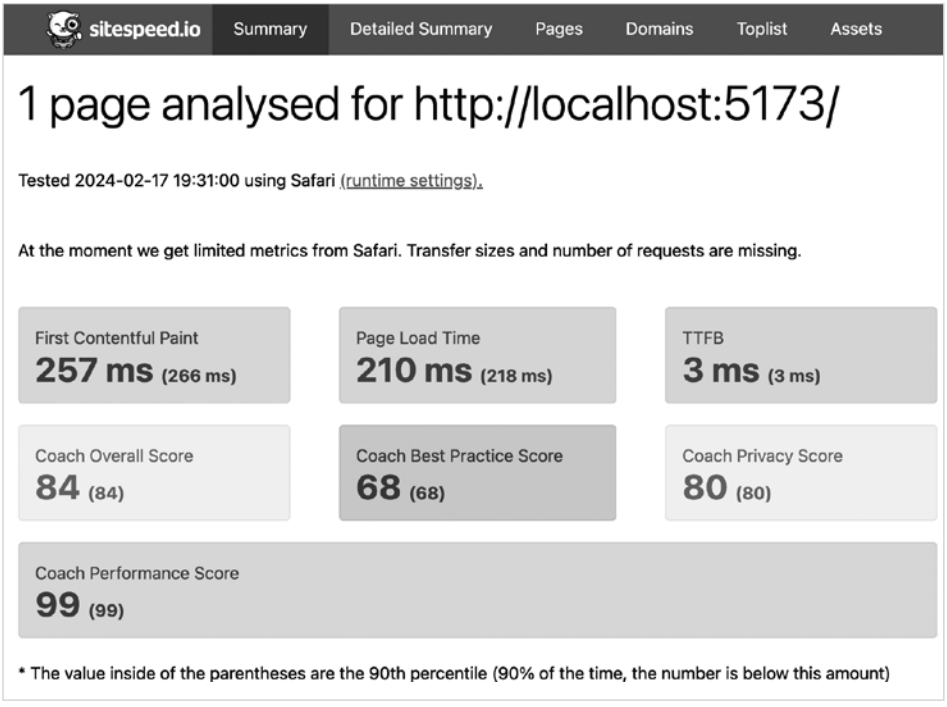

Рис. 20.3. Результаты Sitespeed.io

Нажмите на любой из показателей, чтобы увидеть детали и рекомендации по их улучшению. Для более детального изучения отчетов перейдите в репозиторий проекта на GitHub (https://oreil.ly/Fjn7S) и откройте HTML-файлы в браузере.

Также существуют коммерческие решения для измерения производительности, например Sentry и New Relic. Мы рассматривали их в главе 12. Теперь, после того как вы познакомились с инструментами для анализа сайта, можно заняться улучшением приложения и привлечением команды к этому процессу. Эти задачи можно добавить в процесс ревью пул-реквестов или квартальных аудитов. Некоторые изменения могут кардинально повлиять на процесс разработки, поэтому обсудите их с командой — и не забывайте о демонстрациях.

Направления для улучшения

После выполнения тестов нужно определить, как исправить проблемы с производительностью. Существует несколько способов оптимизации фронтенда. Некоторые можно внедрять постепенно вместе с новыми функциями, другие потребуют более масштабных вмешательств, так как они меняют принцип работы приложения. Начнем с анализа пакетов в проекте.

Анализ размера бандла

Размер бандла — один из неочевидных факторов, замедляющих работу приложения. Хотя почти для любой задачи существует готовый пакет, это не означает, что их нужно использовать бездумно. Каждый установленный пакет увеличивает размер бандла и замедляет загрузку страницы и время отклика. Многие пакеты также зависят от других пакетов, поэтому в вашем бандле могут оказаться вещи, которые вы не используете напрямую.

Один из способов определить, какие пакеты больше всего увеличивают размер бандла, — проанализировать сборку с помощью инструментов вроде source-map-explorer (https://oreil.ly/Oh8Nd) или vite-bundle-visualizer (https://oreil.ly/PjWjs). Эти инструменты анализа помогут выяснить, какие файлы проекта больше всего влияют на размер бандла и где можно что-то урезать. Здесь я использую vite-bundle-visualizer, поскольку в проекте задействован инструмент Vite. На рис. 20.4 показано, как будут выглядеть результаты vite-bundle-visualizer для вашего проекта, если выполнить следующие команды:

```
npm i vite-bundle-visualizer
npx vite-bundle-visualizer
```

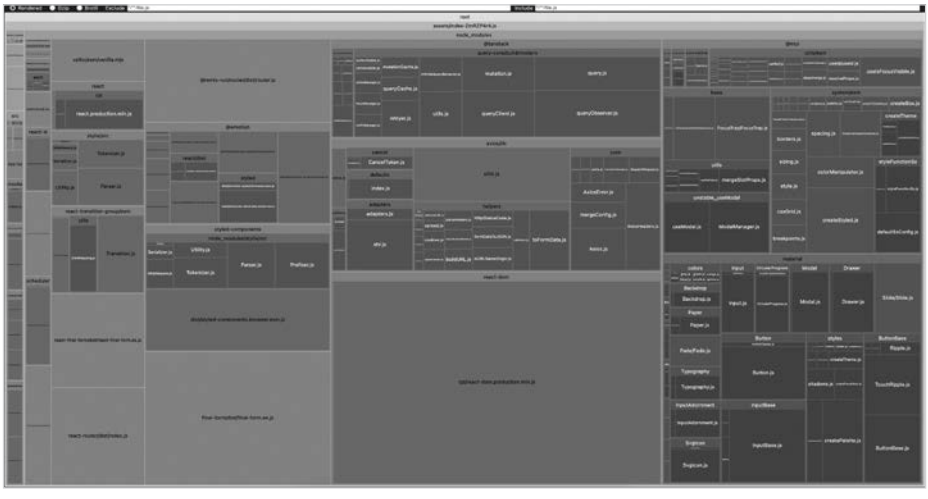

Рис. 20.4. Карта исходного кода для анализа размера бандла с помощью vite-bundle-visualizer

Вы можете кликнуть на каждый из квадратов, чтобы увидеть детали — какой файл больше всего увеличивает размер. Это поможет решить, стоит ли выбрать альтернативный пакет или нужно рефакторить определенные файлы в репозитории.

В одном из проектов, где я работала, использовалась архитектура микрофронтендов (https://oreil.ly/Jrm-H). Разные микрофронтенд-проекты часто раздували основное приложение, и source-map-explorer позволял четко увидеть, какой из них вызывал конкретную проблему. Иногда вы можете с удивлением обнаружить, что размер бандла увеличивается не из-за пакета, а из-за компонента, написанного вами или командой.

После анализа сборки, чтобы понять, где можно уменьшить размер, стоит обратить внимание на саму сборку.

Конфигурации сборки

Тонкая настройка конфигураций сборки — непростая задача, но это важный навык для senior-разработчика, который действительно позволяет проявить себя как сильного специалиста. Уделите время изучению нюансов этой концепции. Иногда улучшение конфигураций сборки помогает уменьшить размер бандла или даже оптимизировать артефакт, который вы разворачиваете в продакшене. Возможно, вам придется работать с проектом на Webpack и обновлять его настройки (https://oreil.ly/29o0o).

К счастью, в этом проекте вы начали с нуля, используя современный инструмент — Vite. Если же вы работаете с реальным веб-приложением на Create React App (CRA), то такой инструмент, как CRACO (https://oreil.ly/CKnaI), поможет получить доступ к настройкам конфигурации, которые улучшают производительность. Поскольку проект написан на TypeScript, содержимое файла `tsconfig.json` (https://oreil.ly/jmaLT) определяет, как будет компилироваться ваше приложение: какие файлы войдут в продакшен-артефакт, где они расположены, как сокращаются или сжимаются, а также с какими браузерами приложение будет наиболее совместимо. Браузеры не работают с TypeScript напрямую, поэтому ваши `.ts`-файлы в конечном счете будут скомпилированы в JavaScript.

Если вы хотите ускорить работу приложения, проверьте конфигурацию на наличие файлов, которые не нужны в продакшене. Тесты, отчеты и любые файлы, используемые исключительно для разработки, скорее всего, можно исключить из финального артефакта. Также стоит проверить, какие модули JavaScript (https://oreil.ly/grMgR) вы используете, поскольку это повлияет на обработку полифилов (https://oreil.ly/JcQh6) (то есть функциональности, не поддерживаемой в некоторых версиях браузеров).

Функциональность JavaScript менялась с годами, что привело к появлению модулей, таких как ES2016 и ES8, которые добавили асинхронные функции (https://oreil.ly/x0S0V), оператор расширения (https://oreil.ly/twb1Q), опциональную цепочку (https://oreil.ly/UygLk) и оператор нулевого слияния (https://oreil.ly/sLRqn). Чтобы

быть в курсе изменений, периодически проверяйте репозиторий TC39 на GitHub (https://oreil.ly/0nNLf) и изучайте новые предложения.

Браузеры не обновляются синхронно с изменениями в JavaScript, поэтому вашему коду могут потребоваться полифилы для работы со старыми версиями браузеров — в зависимости от конфигурации сборки. Это может замедлить приложение, увеличивая объем кода из-за необходимости прописывания дополнительных условий и включения пакетов, необходимых для поддержки определенных версий браузеров.

Конфигурация сборки в целом может быть сложной. Честно говоря, это одна из самых раздражающих частей разработки, потому что не всегда есть однозначные ответы или стратегии. Требуется экспериментировать, чтобы найти оптимальное решение. Главное, сохранять упорство и не сдаваться. Я сама не раз сталкивалась с этим.

Конфигурация кэширования

Пробуя разные стратегии кэширования, можно значительно повысить производительность. Кэширование — это когда вы решаете, какие ресурсы и данные следует сохранять в браузере, чтобы ускорить загрузку. Через определенное время данные в кэше считаются устаревшими, а значит, их нужно обновить с сервера. Вы можете задать этот временной интервал, поэтому важно найти баланс между производительностью (дольше храня данные в кэше) и актуальностью (кэшированные данные могут расходиться с серверными). Шрифты и изображения обычно хранятся в кэше браузера дольше, так как они редко меняются.

Определение оптимального времени кэширования перед каждым новым запросом актуальных данных — это целое искусство. Подумайте, каково оптимальное время для обновления ваших данных. Некоторые, например информация о банковском счете или заказе, должны запрашиваться при каждой загрузке страницы. Другие, такие как списки товаров или календари событий, не требуют частого обновления. Обсудите с командой, какой подход лучше применить.

Сложность кэширования в том, что оно может усложнить отладку. Если у пользователя (или даже у разработчика) данные закэшированы локально, приложение может обращаться к чему-то, чего уже не существует. Если вы столкнулись с ситуацией, когда данные отображаются некорректно, предложите очистить кэш или попробовать другой браузер. Использование устаревших данных из кэша, содержащих значения, которых больше нет, может привести к падению приложения.

Вы можете реализовать любые стратегии (https://oreil.ly/r1A8n) для настройки кэширования на фронтенде. Например, задать время устаревания для определенных эндпоинтов или исключить некоторые эндпоинты из кэширования. Главное, четко документировать, как обрабатывается кэш. Документация может быть простой, например комментарии в коде, чтобы команда разработчиков понимала логику каждого варианта кэширования.

Сейчас доступно множество инструментов, упрощающих эту задачу. Например, React Query (https://oreil.ly/gwO1l) поддерживает кэширование «из коробки», поэтому изучите его стандартные настройки и документацию React (https://oreil.ly/bwiB5) перед внесением изменений. Встроенные хуки React, такие как `useContext` (https://oreil.ly/fMq42), `useMemo` (https://oreil.ly/0FzC8) и `useCallback` (https://oreil.ly/gOfCK), также помогают кэшировать данные. Доступно множество вариантов, поэтому рекомендую изучить документацию.

Также потребуется механизм *сброса кэша* (cache busting) — принудительной перезагрузки страниц или удаления текущих кэшированных данных пользователя. Это может понадобиться, если:

- обновления API изменяют ответы так, что ломают фронтенд;
- требуется срочно разослать пользователям обновленные данные из-за внешних требований.

Реализовать это можно двумя способами:

- написать код для принудительной перезагрузки страниц с временным изменением стратегии кэширования;
- использовать CDN (*Content Delivery Network*) (https://oreil.ly/3Cm0C) для централизованного распространения изменений.

CDN ускоряют работу приложения, храня копии контента приложения на серверах, которые находятся ближе к пользователям. Однако это усложняет отладку в продакшене: если проблемы при применении CDN возникают лишь у части пользователей, вероятно, причина в кэше. Решение: очистить кэш через панель управления или CLI провайдера CDN и посмотреть, поможет ли это.

В React Query кэширование настраивается параметрами запросов к эндпоинтам. Пример настройки времени устаревания (staleTime) для очистки кэша и повторного запроса данных:

```
const {
  isLoading: ordersIsLoading,
  error: ordersErrors,
  data: orderData,
} = useQuery({
  // Данные хранятся в кэше 1 час (в миллисекундах), затем удаляются сборщиком
мусора
  gcTime: 3600000,
  // Повторный запрос данных через 1 час (в миллисекундах)
  refetchInterval: 3600000,
  // Данные считаются устаревшими через 30 минут (в миллисекундах)
  staleTime: 1800000,
  queryKey: ['orderData'],
  queryFn: () =>
```

```
    axios.get(`${import.meta.env.API_URL}/v1/orders`).then((res) => res.data),
})
```

Главное, что вам и команде нужно понять, — как параметры конфигурации фактически влияют на ваши данные. В этом случае стоит посмотреть документацию (https://oreil.ly/se6sB) и детально разобраться в используемом пакете. После внедрения стратегии кэширования пользователи заметят разницу в работе приложения. Не забывайте только учитывать кэш при отладке проблем.

Ленивая загрузка (Lazy Loading)

Ленивая загрузка ускоряет время загрузки страницы благодаря первоочередной загрузке важного контента, а остального — по мере необходимости. Таким образом, пользователю не приходится ждать полной загрузки страницы, чтобы увидеть хоть что-то. Этот метод применим на страницах с функционалом загрузки при прокрутке.

По мере необходимости контент подгружается, и в эти моменты пользователь видит индикатор загрузки. Такая подгрузка может значительно повлиять на показатель CLS (Cumulative Layout Shift), поскольку некорректное оформление загружаемых блоков способно вызывать резкие изменения размеров страницы.

К счастью, в React есть встроенные решения: компонент `<Suspense />` (https://oreil.ly/uVQmO) или метод `lazy` (https://oreil.ly/gZWtk). Они позволяют задать состояние загрузки для компонентов, зависящих от данных, гибко реализовать загрузку при прокрутке и использовать пользовательские компоненты загрузки для разных элементов страницы. Такие компоненты иногда называют «скелетами», так как они соответствуют размерам компонентов, которые в итоге загрузятся.

Например, можно стилизовать `<div />` под таблицу заказов в вашем приложении. Решение не обязательно должно быть сложным, но оно поможет избежать проблемы смещения контента (content layout shift). Вот как это может выглядеть при реализации с компонентом `<Suspense />`:

```
…
const TableLoadingSkeleton = () => (
  <table>
    <thead>
      <tr>
        <td>Товар</td>
        <td>Описание</td>
        <td>Цена</td>
      </tr>
    </thead>
    <tbody>
      <tr>
        <td colSpan={3}>Загрузка...</td>
      </tr>
    </tbody>
```

```
  </table>
);
…
<Suspense
  fallback={
    <TableLoadingSkeleton data-testid="orders-loading-circle" />
  }
>
  <OrdersTable orders={orders} />
</Suspense>
…
```

На рис. 20.5 показана итоговая страница.

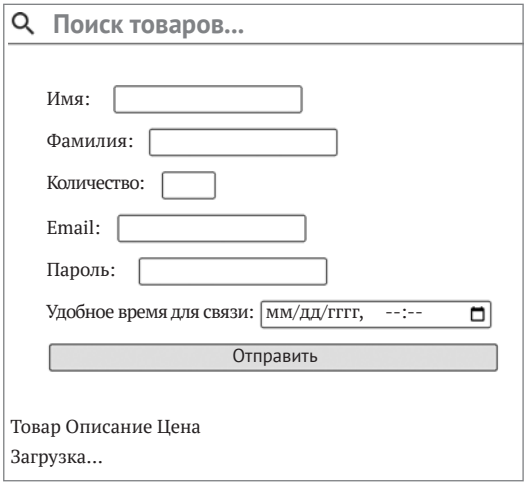

Рис. 20.5. Компонент скелетной загрузки (Skeleton Loading) в `fallback Suspense`

Теперь при задержке загрузки компонента можно явно указать для него собственный компонент загрузки. Важно сохранять хотя бы высоту и ширину компонента загрузки примерно такими же, как у реального компонента. Дополнительное преимущество этого подхода состоит в том, что вы можете обнаружить способы уменьшить размер компонентов, что со временем упростит управление кодом и даже улучшит тестируемость.

Условный рендеринг компонентов на основе состояния загрузки — альтернатива «ленивой» загрузке, но этот метод подходит, если нужно сохранить определенный формат кода. Такое решение стоит обсудить с командой, поскольку это дело вкуса.

Также можно попробовать «ленивую» загрузку пакетов или импортируемых компонентов — это сократит объем загружаемого контента до момента его

фактического использования на странице. Пример ленивого импорта компонента `Header` в файле `UserInfo.Container`:

```
…
const Header = lazy(() => import('../../components/Header'));
…
<Suspense
  fallback={
    <CircularProgress
      color="secondary"
      size={100}
      data-testid="header-loading-circle"
    />
  }
>
  <Header
    userName={userInfo.name}
    joinedDate={userInfo.joinedDate}
    onSubmitSearch={onSubmitProductSearch}
  />
</Suspense>
…
```

Эти способы помогут одновременно улучшить время загрузки страницы и показатель CLS.

Предварительная загрузка

Предварительная загрузка (*Prefetching*) означает получение и сохранение в кэше данных или компонентов, которые, как ожидается, понадобятся пользователю.

React Query упрощает эту задачу с помощью метода `prefetchQuery` (https://oreil.ly/68Nje). *Серверный рендеринг* (Server-Side Rendering, SSR) (https://oreil.ly/XZPYj) также связан с предварительной загрузкой данных, поскольку позволяет загружать целые страницы с сервера при использовании Next.js или React Server Components (https://oreil.ly/yjn1-) — относительно новых возможностей фреймворка.

SSR представляет собой принципиально иную парадигму по сравнению с текущими фронтенд-решениями, и не каждое приложение будет его использовать. Рекомендую изучить документацию по SSR (https://oreil.ly/c3WWl) перед внедрением. Это эффективный инструмент, который может значительно ускорить приложение, но у него есть недостатки: например, компоненты, отрендеренные на сервере, никогда не перерисовываются. Если этого не учесть, отладка может оказаться сложной. Эти компоненты не обновляются при вызове хуков `useState` или `useEffect`, и код, скорее всего, не скомпилируется. Преимущество серверных компонентов в том, что функции для бэкенда (например, запросы к базе данных) можно писать так же, как и для фронтенда, не раскрывая их пользователям.

SSR полезен для статического контента, который не меняется при загрузке страницы, или для функциональности, не зависящей от пользователя (например,

общие календари событий или списки товаров). Производительность может быть ограниченна, если весь контент пользовательский (например, данные аккаунта или дашборды). Изучите документацию (https://oreil.ly/4_f-3), чтобы составить собственное мнение. У каждой технологии есть свои сценарии применения — важно понимать, когда и почему ее использовать.

CSS, изображения и шрифты

В завершение главы рассмотрим оптимизацию ресурсов. Изображения особенно сильно влияют на скорость загрузки страниц из-за их размера и времени скачивания.

Оптимизация изображений, — например их сжатие, использование изображений с сервера в правильных размерах и применение форматов вроде SVG, может ускорить их загрузку. Также существуют форматы WebP (https://oreil.ly/QKgCo) и AVIF (https://oreil.ly/rkc4P), но пока они не поддерживаются всеми браузерами, поэтому может потребоваться вернуться к SVG или PNG. Проверить поддержку браузерами можно на сайте CanIUse (http://caniuse.com).

Что касается CSS-файлов, независимо от того, используете вы готовый пакет или собственные стили, их нужно минимизировать и удалять из них неиспользуемые правила, из-за которых разрастается бандл и замедляется загрузка страницы. CSS может справиться со многими задачами, которые обычно решают с помощью JavaScript: например, темная тема, анимация градиентов и параллакс-эффекты. Прежде чем прибегать к JavaScript, изучите последние возможности CSS (https://oreil.ly/NQSMl), так как этот язык постоянно развивается, особенно в области адаптивного дизайна и анимации.

Разработчики часто упускают из виду шрифты, так как их обычно загружают из сторонних сервисов, например Google Fonts CDN (https://fonts.google.com). Убедитесь, что загружаете только необходимые шрифты и удаляете неиспользуемые. Если возможно, размещайте шрифты на собственном сервере — это ускорит загрузку страниц, так как не придется зависеть от стороннего CDN. То же касается библиотек вроде MUI: импортируйте только нужные компоненты, а не всю библиотеку целиком, чтобы уменьшить объем загружаемого контента.

Заключение

В этой главе вы узнали, как тестировать производительность приложения и находить точки роста. Помните: низкая производительность отсекает часть пользователей. Такими пользователями могут быть жители сельской местности, регионов с медленным интернетом, те, кто использует только мобильные телефоны, старое оборудование или тарифы с оплатой за мобильный трафик. Часто мы, разработчики, привыкшие к мощному железу и быстрому интернету, упускаем это из виду. Но не забывайте: не у всех есть такие возможности.

Чем больше вы узнаете о взаимодействии пользователей с приложением, тем лучше сможете принимать решения вместе с командой на основе данных. В идеале следует учитывать лучшие практики на ранних этапах разработки, но предугадать действия пользователей невозможно, пока они не увидят приложение.

Отслеживайте ключевые метрики производительности в динамике — это поможет понять поведение пользователей и сосредоточиться на действительно важных аспектах. Оптимальные подходы к поддержке браузеров и работе с кэшем проявятся со временем. На старте делайте все возможное, но и дальше смело вносите изменения, если по метрикам выходит, что возникла такая необходимость. Приоритеты пользователей тоже будут меняться. Главное, определить ключевые направления для улучшения, обсуждать их с командой и корректировать решения на протяжении жизненного цикла продукта.

ГЛАВА 21

Тестирование фронтенда

В предыдущих главах я упоминала, что лучшая практика — писать тесты параллельно с подготовкой новой функциональности или рефакторингом.

Тестирование заслуживает отдельного внимания, и здесь мы подробно его разберем. При разработке приложения важно избегать регрессий в имеющейся функциональности. *Регрессия* — это когда новый код непреднамеренно вызывает ошибки в работающих частях приложения. Команда QA (если она есть) физически не успеет проводить регрессионное тестирование для каждого релиза, поэтому разработчики должны действовать на опережение и самостоятельно обеспечивать надежность своего кода. Тесты для нового кода часто выявляют проблемы функциональности и показывают, что происходит, когда есть ошибки.

В этой главе мы рассмотрим:

- методы определения того, какие части фронтенда тестировать;
- модульное тестирование;
- сквозное (end-to-end, e2e) тестирование;
- полезные инструменты для тестирования.

Две главные цели написания тестов: предотвращение попадания багов к пользователям и документирование поведения приложения. Тесты также повышают уверенность при дальнейшей разработке, снижая риск неожиданных поломок.

Совместная работа над тест-кейсами сплачивает команды разработки, продукта, дизайна и QA. Важно помнить: команда QA — не враг. Ее работа критична для уверенного развертывания. Она не критикует ваш код, а лишь указывает на расхождения с требованиями к функционалу.

Определение тестовых сценариев

Если вы не уверены, с чего начать, внимательно изучите свои компоненты. Вот список точек, где нужны тесты:

- точки условного рендеринга (conditional rendering);
- точки выполнения API-запросов;

- точки возможных ошибок;
- точки обработки данных;
- практически любой элемент, влияющий на рендеринг компонента.

Нередко один компонент имеет объемный файл тестов, особенно если его функциональность меняется. При ревью пул-реквестов проверяйте наличие новых тест-кейсов, чтобы обеспечить максимально возможное покрытие кода (test coverage). Когда продуктовая команда предоставляет требования, сразу продумывайте тест-сценарии.

Однако здесь важен баланс: некоторые функции требуют больше времени на тестирование, чем на реализацию. В таких случаях оценивайте компромиссы между усилиями на написание тестов, сроками релиза и временем разработки. Лично я никогда не видела фронтенд-приложения с 100 %-ным покрытием тестами — в отличие от бэкенда, где встречаются проекты с такой высокой степенью покрытия.

Это связано со сложностью инструментов: тесты сложно адаптировать под рендеринг из-за нестабильности таймингов, событий и зависимостей от сторонних сервисов. Со временем вы научитесь определять, когда тест отнимает больше времени, чем приносит пользы. Жестких правил нет — полагайтесь на опыт, интуицию и командную согласованность, чтобы никто не застрял на пул-реквесте из-за сложного теста.

Модульные тесты

Не важно, используете ли вы BDD (https://oreil.ly/acx5Y) или TDD (https://oreil.ly/7l5l2): если код тестируется, у него хорошее покрытие тестами и тесты проходят — этого достаточно. *Модульные тесты* — основа фронтенд-тестирования. Они программно проверяют работу компонентов, включая граничные случаи, и выявляют проблемы вроде условного рендеринга. Такие тесты гарантируют стабильность кода при каждом развертывании и повышают уверенность команды при частых релизах.

Jest и React Testing Library

Если вам нужна более зрелая библиотека для работы, Jest (https://jestjs.io) и React Testing Library (https://oreil.ly/VXJto) (RTL) — надежные варианты с обширной документацией, примерами и поддержкой сообщества. Вы уже использовали Jest для тестирования бэкенда: это базовый фреймворк для тестового набора NestJS. Однако его применение во фронтенде отличается. Здесь пригодится RTL.

Проекты, инициализированные с помощью CRA (https://create-react-app.dev), включают RTL «из коробки». Однако команда React объявила CRA (https://oreil.ly/6bFGa) устаревшим в начале 2023 года, поэтому этот проект был создан с помощью Vite. Хотя RTL не используется по умолчанию во всех альтернативных решениях для CRA, его все равно полезно добавить в проект. Способ написания тестов почти не

будет отличаться от того, который вы используете с инструментом, применяемым в этом проекте.

Vitest

Поскольку проект инициализирован с помощью Vite, для модульных тестов мы будем использовать Vitest (https://vitest.dev). Он совместим с Jest, поэтому переход будет простым даже для команды, знакомой с Jest и RTL. Это новый инструмент, который быстро набирает популярность в экосистеме React. По сравнению с Jest он может значительно ускорить выполнение всего набора тестов.

При миграции с CRA на Vite или другой инструмент проверяйте бенчмарки скорости выполнения. Вы заметите разницу в скорости тестов, времени «холодного старта» приложения и, возможно, даже во времени отклика. Эти различия особенно важны при развертывании приложения в разных средах. Мы подробно рассмотрим пайплайны развертывания в главе 27, но, поскольку вы работаете с ними сейчас, стоит обратить внимание на время выполнения модульных тестов. Быстрые тесты также ускорят локальную разработку, поэтому проанализируйте доступные варианты и выберите оптимальный.

Давайте напишем несколько тестов для компонента `UserInfo`, чтобы создать примеры для команды в вашем репозитории. Это даст вам представление о том, как можно разбить компонент на различные тестовые сценарии. Сначала подготовим моки для тестов:

```
import { afterEach, beforeEach, describe, expect, it, vi } from 'vitest';
import { act, render, screen } from '@testing-library/react';
import UserInfo from '.';
import { orderResponseData } from '../../mocks/orders';
import { userResponseData } from '../../mocks/users';

const mocks = {
  useQuery: vi.fn(),
  useErrorBoundary: vi.fn(),
  showBoundary: vi.fn(),
};

vi.mock('@tanstack/react-query', () => ({
  useQuery: () => mocks.useQuery(),
}));

vi.mock('react-error-boundary', () => ({
  useErrorBoundary: () => mocks.useErrorBoundary(),
}));

describe('<UserInfo />', () => {
  beforeEach(() => {
    mocks.useQuery.mockReturnValue({
      isLoading: false,
      data: orderResponseData,
```

```
    });
    mocks.useQuery.mockReturnValue({
      isLoading: false,
      data: userResponseData,
    });
    mocks.useErrorBoundary.mockReturnValue({ showBoundary: mocks.showBoundary });
  });

  afterEach(() => {
    vi.clearAllMocks();
  });
});
```

Настройка всех моков для тестов упростит их написание при работе со сложными сценариями. В этом примере вы настроили моки для хука `useQuery`, который получает данные, а также моки для функций обработки error boundary. Эти моки позволят возвращать различные значения или ответы при написании тестов без необходимости вызывать реальные методы в компоненте. Это важно, чтобы не тестировать функциональность вызываемого метода, — можно сосредоточиться на работе кода.

Если проверить импорты, вы найдете `orderResponseData` и `userResponseData` в отдельных файлах. Это мок-ответы, определенные для API-запросов, выполняемых в компоненте. Они вынесены в отдельные файлы, потому что могут использоваться в других тестовых файлах для разных компонентов. Хранение их в отдельных файлах повышает поддерживаемость, так как можно обновлять один источник вместо дублирования данных в нескольких файлах.

Внутри блока `describe` инициализируйте моки в состоянии, ожидаемом компонентом. Это включает выполнение API-запросов, возврат мок-данных в качестве ответов и установку других значений. Это делается в блоке `beforeEach`, который выполняется перед каждым тестом в `describe`. В какой-то момент вы измените ответы или данные, и они не должны влиять на следующий тест. Поэтому принято сбрасывать значения в `afterEach`.

Далее переходим к тест-кейсам для проверки состояний рендеринга. Хорошей практикой является наличие хотя бы одного теста, который корректно рендерит компонент. При выборе значений для проверки убедитесь, что они отражают то, что должно отображаться на основе данных или действий пользователя. Тестирование статического текста может вводить в заблуждение, так как он не меняется в зависимости от состояния компонента. Вот пара тестов для этого компонента внутри `describe`, с которых можно начать:

```
it('должен отображать экран информации о пользователе', async () => {
  render(<UserInfo />);

  expect(screen.getByPlaceholderText('Поиск товаров...')).toBeDefined();
  expect(screen.getByText('Товары для собак')).toBeDefined();
  expect(screen.getByText('Клиент с 2002 года')).toBeDefined();
```

```
});

it('должен отображать индикатор загрузки, пока загружаются данные пользователя
', async () => {
  mocks.useQuery.mockReturnValue({
    isLoading: true,
  });

  render(<UserInfo />);

  expect(screen.getByTestId('user-loading-circle')).toBeDefined();
});
```

В первом тесте видно, что компонент рендерится, после чего проверяются значения, полученные из API-запросов, а также один из интерактивных компонентов. Если немного изменить значения, тест провалится. Именно так можно убедиться, работает он или нет. Второй тест проверяет отображение спиннера загрузки во время ожидания ответа от API.

Чтобы убедиться, что все сценарии тестирования охвачены, я держу файл компонента и теста рядом на разных экранах. Это позволяет построчно сопоставлять код и добавлять соответствующие проверки в тест. Такой подход помогает глубже понимать функционал и разбираться в сложной логике рендеринга.

Еще один тип тестов, который стоит рассмотреть, — снапшот-тесты (https://oreil.ly/MdzF1). Они сравнивают вывод вашего кода с заранее сохраненным снапшотом, содержащим точное ожидаемое HTML-представление или значения. Это полезно для контроля неожиданных изменений UI или подтверждения обновлений. Снапшот-тесты (https://oreil.ly/MdzF1) обычно легко поддерживать, так как они не требуют написания кода: достаточно выполнить команду (https://oreil.ly/Le0Xr), когда необходимо обновить снапшот-файл.

После погружения в детали написания тестов часто становятся заметны участки кода, которые можно сделать более понятными. Рефакторинг для улучшения тестируемости — это хорошая практика. Вы можете показать команде, где код можно упростить или вынести во вспомогательные функции/хуки, повысив частоту его переиспользования и удобство поддержки.

Иногда сложность написания теста указывает на запутанность реализации. Модульные тесты помогут вам выявить проблемные места и улучшить их. Вот пример кода, который довольно сложно тестировать, несмотря на простоту реализации:

```
const updateDate = () => {
  const now = new Date()

  const formattedDate = new Intl.DateTimeFormat('en-US').format(now)

  return formattedDate
}
```

При тестировании этого кода возникают сложности, поскольку каждый раз используется текущая дата. Это может привести к нестабильным тестам, которые будут падать из-за расхождений в миллисекундах. Вот как можно рефакторить этот код, чтобы упростить тестирование и добиться стабильных результатов:

```
const updateDate = (date?: string) => {
  const now = date ? new Date(date) : new Date()

  const formattedDate = new Intl.DateTimeFormat('en-US').format(now)

  return formattedDate
}
```

Теперь вы можете передавать конкретную дату для получения воспроизводимого результата. Когда у вас и у команды войдет в привычку писать юнит-тесты для всего нового функционала, вы обнаружите, что это помогает выявлять регрессии на этапе разработки и заставляет задуматься о том, как лучше структурировать код.

Mock Service Worker

Существует множество инструментов тестирования, которые могут оптимизировать процессы как внутри команды, так и на других этапах жизненного цикла разработки. Один из таких инструментов — MSW, Mock Service Worker (https://oreil.ly/H6Y9g), который позволяет имитировать эндпоинты. Когда нужно убедиться, что эндпоинт вызывается с правильными параметрами и возвращает ожидаемые данные, моки данных крайне полезны, и MSW выводит этот процесс на качественно иной уровень. Это особенно ценно, если требуется развернуть приложение в среде разработки или в функциональной среде, чтобы протестировать или чтобы продуктовая команда могла изучить работу приложения в различных сценариях. Также стоит рассмотреть альтернативы вроде Nock (https://oreil.ly/1eKG7) или JSON Server (https://oreil.ly/oOABB).

Я видела, как инструменты вроде MSW (Mock Service Worker) используют в организациях, где бэкенд и фронтенд — это отдельные команды: фронтенд-команда работает, используя MSW на основе ожидаемых данных. Однако здесь нужно быть осторожным. Иногда фронтенд могут начать разрабатывать до того, как бэкенд будет готов, и это считается плохой практикой, если только вы не находитесь в постоянном тесном контакте с бэкенд-командой. Такой подход может привести к избыточной работе, если бэкенд-команда внесет изменения в структуру данных.

MSW работает, создавая сервис-воркер (service worker) в вашей публичной директории. Затем вы оборачиваете компонент `<App />` в функцию (https://oreil.ly/5hfUQ), чтобы определить, будет ли приложение использовать реальные API или моки, которые вы создадите. Следовательно, файл `main.tsx` в вашем проекте будет выглядеть так: (https://oreil.ly/5hfUQ).

```
export async function shouldEnableMocking() {
  if (import.meta.env.MODE !== 'development') {
    return;
  }

  const { worker } = await import('./mocks/browser');

  // `worker.start()` возвращает Promise, который разрешается,
  // когда Service Worker запущен и готов перехватывать запросы.
  return worker.start();
}

shouldEnableMocking();

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

Теперь при запуске приложения в режиме разработки вы будете получать ответы от MSW. Для завершения реализации этого инструмента осталось только создать обработчики, которые будут вызываться вместо реального API. В директории src/mocks создайте новый файл `handlers.ts` и добавьте в него следующий код:

```
import { http, HttpResponse } from 'msw';
import { userResponseData } from './users';
import { orderResponseData } from './orders';

export const handlers = [
  http.get(`${import.meta.env.VITE_API_URL}/v1/users`, () => {
    return HttpResponse.json(userResponseData);
  }),
  http.get(`${import.meta.env.VITE_API_URL}/v1/orders`, () => {
    return HttpResponse.json(orderResponseData);
  }),
];
```

Вот так можно настроить мок-эндпоинты и их ответы (https://oreil.ly/esMrR). Полагаю, теперь стало понятно, насколько удобно хранить мок-ответы в отдельных файлах? Отсюда можно добавлять новые эндпоинты и условия для обработки токенов авторизации и параметров.

Сквозное тестирование с Cypress

Последний тип тестирования, который я рассмотрю, — это *сквозные тесты* (e2e), позволяющие автоматизированно проверять полный сценарий взаимодействия пользователя с приложением. С помощью инструмента типа Cypress (https://oreil.ly/-McxR) можно запускать автоматизированные тесты с точки зрения пользователя.

Cypress открывает собственный браузер и выполняет действия так, как это сделал бы пользователь. Он нажимает кнопки, вводит значения в формы и проверяет, появляются ли ожидаемые изменения на странице.

Совместно работайте с продуктовой командой, чтобы получить все сценарии. Использование инструментов вроде Cucumber (https://oreil.ly/AoLzZ) и Gherkin (https://oreil.ly/p1wmc) поможет продуктовой команде понять цель тестирования и активнее участвовать в процессе. Gherkin и Cucumber позволяют писать тестовые сценарии в формате `given-when-then`, что помогает продуктологам точнее формулировать требования, пока вы вместе прорабатываете спецификации и моки.

В зависимости от организации сквозные тесты могут быть задачей команды QA. Это особенно актуально, если в команде есть *инженеры по тестированию* (SDET, software development engineer in test) — специалисты, которые занимаются написанием автоматизированных тестовых сценариев с использованием инструментов типа Cypress. SDET обычно применяются в крупных компаниях, хотя иногда их можно встретить и в стартапах.

Вот пример теста на Cypress:

```
describe('Информация о пользователе', () => {
  it('Переход на главную страницу TestStore', () => {
    cy.visit('http://localhost:5173/')
    cy.contains('Действия')

    cy.contains('Действия').click()
    cy.url().should('include', '/actions')
  })
})
```

При запуске этого теста в Cypress вы увидите процесс точно так же, как его видит пользователь. Инструмент запустится в отдельном окне, как показано на рис. 21.1.

Теперь рассмотрим пример с использованием Gherkin и Cucumber. Можно попросить продуктовую команду писать требования в формате Gherkin:

```
// cypress > e2e > user-info.feature
Feature: Функциональность информации о пользователе

  Scenario: Переход на экран действий пользователя
    Given Я нахожусь на экране информации о пользователе
    When Я кликаю по ссылке навигации "Действия"
    Then Я должен быть перенаправлен на экран действий пользователя
```

Этот код будет находиться в файле `user-info.feature` в папке cypress/e2e. На основе этой фичи можно использовать ключевые слова `Given...When...Then` для определения шагов в сквозном тесте, создав файл `user-info.ts` в той же папке. Код в `user-info.ts` будет выглядеть так:

```
// cypress > e2e > user-info.ts
import { When, Then, Given } from '@badeball/cypress-cucumber-preprocessor'

Given('Я нахожусь на экране информации о пользователе', () => {
  cy.visit('http://localhost:5173/')
})

When('Я кликаю по ссылке навигации "Действия"', () => {
  cy.contains('Действия')
  cy.contains('Действия').click()
})

Then('Я должен быть перенаправлен на экран действий пользователя', () => {
  cy.url().should('include', '/actions')
})
```

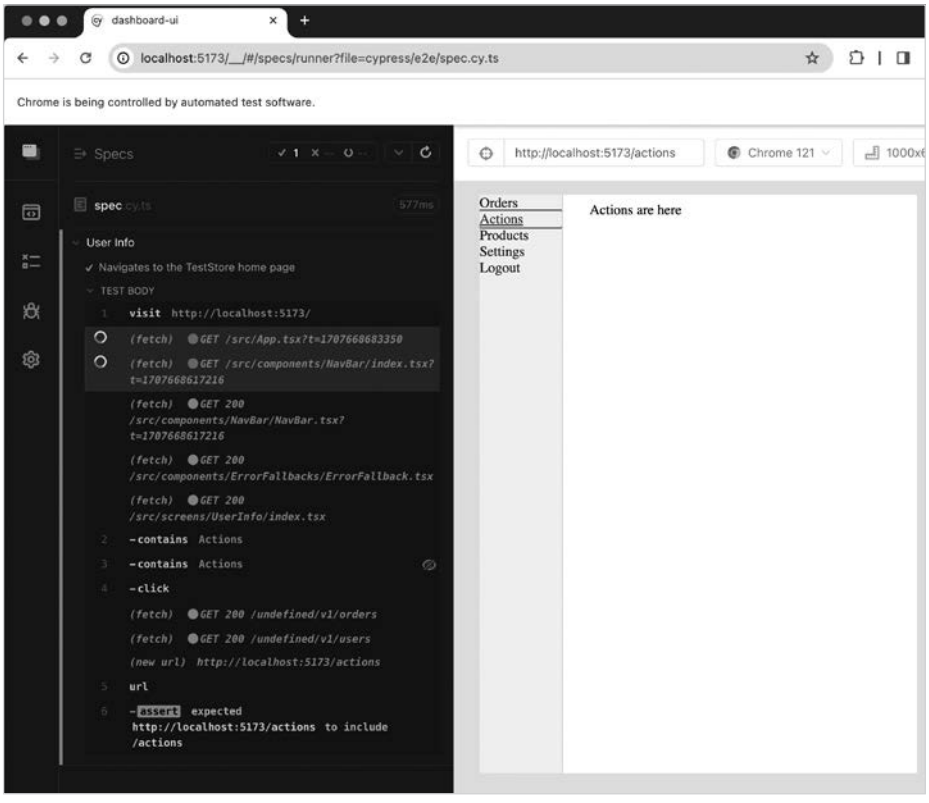

Рис. 21.1. Запуск теста Cypress в отдельном окне

Вот мы и посмотрели несколько способов реализации сквозных тестов. Вам не обязательно использовать Gherkin и Cucumber для максимальной эффективности Cypress, но эти инструменты могут быть полезны, если команды посчитают их

удобными. Некоторые разработчики утверждают, что Gherkin и Cucumber *усложняют* написание и поддержку тестов в Cypress, поскольку Cypress и так достаточно хорош «из коробки»; однако в корпоративной среде эти инструменты позволят привлекать продуктовую команду к техническим обсуждениям.

Здесь нет однозначно правильного или ошибочного подхода. Все зависит от того, какой вариант вы и ваша команда, совместно с другими командами и руководством, сочтете оптимальным. Лично я предпочитаю писать сквозные тесты напрямую в Cypress, поскольку настройка Gherkin и Cucumber для работы в пайплайне развертывания может оказаться нетривиальной из-за конфигураций.

Заключение

В этой главе мы познакомились с различными способами тестирования фронтенд-приложения. Иногда важно добиваться включения технических задач в требования к функционалу — это навык, который вам пригодится. Тесты относятся к таким задачам, даже в условиях жестких сроков. Если только продакшен не «горит» и вам не нужно срочно выпускать горячее обновление, аргументируйте необходимость написания тестов как часть постмортем-чек-листа. (Мы разбирали это в главе 12.)

Есть соблазн сделать тикеты и пообещать вернуться к ним позже, но имейте в виду, что времени никогда не хватает и часто к ним так и не возвращаются. Не все организации или разработчики одинаково ценят тесты, поэтому поощряйте эту практику с самого начала. Даже если вы работаете с уже готовым приложением, частичное покрытие тестами лучше, чем полное их отсутствие. Тесты не ускоряют разработку новых функций, но определенно помогают поддерживать стабильность кода с технической точки зрения.

ГЛАВА 22

Отладка фронтенда

Наконец, код вашего фронтенда готов к продакшену. В процессе разработки вам уже приходилось отлаживать различные ошибки, поэтому у вас уже есть определенные стратегии и подходы. Теперь, когда все интегрировано и приложение работает в продакшене, вам понадобится более системный метод поиска и исправления проблем.

Хотя и не существует универсальной стратегии, которая гарантированно поможет находить баги одинаковым способом каждый раз, есть общие методики, которые послужат хорошей отправной точкой. Когда вы и другие команды начнете совместно работать с приложением в продакшене, ваше умение быстро выявлять первопричины проблем придется очень кстати.

В этой главе мы разберемся:

- с чего начать отладку;
- какие инструменты могут помочь;
- на что обратить внимание, если вы зашли в тупик;
- что делать, если ошибка вызвана действиями другой команды.

Отладка — это искусство, для совершенствования в нем требуются время и работа над разными проектами. Это один из самых недооцененных, но крайне ценных навыков, поскольку он применим к любым приложениям и языкам программирования. Даже если мы стараемся избегать ошибок в коде, они все равно появляются. Чем больше причин багов вы встречаете в работе, тем больше методов отладки сможете добавить в свой арсенал.

Процесс отладки

Первое, что нужно сделать, — это найти баг. Специалист из команды поддержки может сообщить о нем из-за большого количества обращений пользователей, а вы — обнаружить проблему в продакшене с помощью инструментов мониторинга и оповещений. Возможно, во время отладки другой задачи вы заметите повторяющиеся ошибки в логах. Также есть вероятность, что вы, кто-то из команды разработчиков или QA-инженер заметите аномалию во время тестирования других функций.

После подтверждения наличия бага в продакшене начните собирать информацию из различных источников. Сбор информации включает в себя:

- обсуждение с командой поддержки этапов воспроизведения багов, которые видят пользователи;
- детальный анализ логов;
- изучение последних изменений, развернутых в продакшене.

Как только соберете достаточно информации для воспроизведения бага, переходите к следующему шагу: определению способа исправления и написанию тестов.

Во многих случаях исправление может быть однострочным или минимальным изменением кода. Однако именно поиск места, где нужно внести это изменение, занимает больше всего времени. После внесения исправления завершите написание тест-кейсов, чтобы обеспечить покрытие этого сценария. Наконец, разверните исправление в продакшене и убедитесь, что оно работает корректно.

Это стандартный процесс отладки, который можно применять. Далее мы рассмотрим, как обнаружить и воспроизвести проблему после ее выявления.

Анализ логов

После обнаружения ошибки анализ логов — один из первых шагов для поиска первопричины. С помощью логов можно выявить триггеры ошибок и найти соответствующие записи. Также стоит проверить логи бэкенда — возможно, ошибки API стали причиной проблем во фронтенде.

Выбранный инструмент логирования должен поддерживать функциональность поиска по журналам событий. Если вам известно, на какой странице произошла ошибка, то можно искать по конкретной странице.

Поскольку объем логов может быть большим, важно сужать область поиска. Сосредоточьтесь на:

- повторяющихся шаблонах ошибок;
- событиях, которые стабильно приводят к сбоям;
- данных, которые фиксируются в момент ошибки;
- пользователях, столкнувшихся с проблемой.

Это область, где вы действительно можете проявить себя, поскольку анализ логов бывает сложной задачей. Имейте в виду, что дополнительные логи от приложения будут полезны, — смело добавляйте их в код и развертывайте изменения! Логи бэкенда в любом случае должны быть недоступны для внешних сторон из соображений безопасности, поэтому вы можете записывать больше кастомных данных, чтобы получить детальную информацию о происходящем. Если ошибка связана

с API-запросом, добавьте больше логов на бэкенде, чтобы проверить, получает ли он запрос и ожидаемый ответ.

Логи могут показать, передал ли компонент корректные параметры запроса. Вы можете сделать сообщения в логах более конкретными, например, добавив токен аутентификации пользователя для проверки его прав доступа. Или, если сообщения об ошибках отображаются не полностью, возможно, потребуется обновить конкретный логгер. Например, если в логе указано что-то вроде `error from orders.createOrder`, это не объясняет, что именно произошло. Такое сообщение также может означать, что основная ошибка подавляется кастомной обработкой ошибок, реализованной вами или командой на фронтенде или бэкенде. Изучите код, найдите это сообщение об ошибке и проверьте настройки логгера, чтобы понять причину.

При анализе логов важно учитывать, как обрабатываются объекты: если в сообщениях встречается `[object Object]`, скорее всего, данные нужно преобразовать в строку с помощью `JSON.stringify()`.

Это особенно актуально для ошибок сторонних сервисов, например, при прямом вызове функций таких сервисов, как Okta (https://oreil.ly/FnSom), на фронтенде. Хотя вы не контролируете информацию, которую возвращают ошибки этих пакетов, можно передавать ее в исходном формате — это поможет отследить баги до настроек в дашборде сервиса.

При локальном воспроизведении ошибки не стесняйтесь добавлять логгеры по всему коду для отслеживания сложных ошибок: если логгеры загромождают код, их всегда можно удалить после обнаружения проблемы. Учтите, что некоторые ошибки могут не воспроизводиться локально из-за отсутствия подключения к определенным сервисам или особенностей конфигурации окружения.

Логировать можно не только ошибки — используйте уровни debug, info и warn, как обсуждалось в главе 9. Это поможет выявить шаблоны поведения приложения и отклонения от нормы. Проявите инициативу и создайте для команды чек-лист анализа логов, который можно дополнять по мере накопления опыта.

Вот мой небольшой чек-лист для поиска ошибок в логах:

- определить время возникновения ошибки и отфильтровать логи по этому периоду;
- найти сообщения, связанные с проблемой (страница, компонент, вызов API, ID пользователя);
- изучить полный объект лога, включая все детали;
- проанализировать стек вызовов, если он присутствует, — это поможет локализовать файл с ошибкой;
- проверить смежные логи — записи до и после инцидента могут дать важный контекст;

- перейти в код и найти проблемную область;
- если причина неясна, добавить дополнительные логи в проблемной области и перепроверить вывод.

Для трассировки ошибки по стеку вызовов необходимо правильно настроить карты исходного кода (source maps, https://oreil.ly/K5RIL). Во многих современных инструментах, таких как Vite (https://oreil.ly/K8eSN), Rollup (https://oreil.ly/rYeih) и esbuild (https://oreil.ly/ybP_o), эта функция по умолчанию отключена. Для ее включения требуется задать специальный параметр в конфигурации.

Это лишь первый шаг. Анализируя логи приложения, отслеживайте цепочку событий — это важно. Не существует строгих правил, какая именно информация должна попадать в логи при ошибках, — более того, некоторые ошибки могут вообще не логироваться. Однако четкое понимание того, что вы ищете, поможет быстрее разобраться в проблеме и перейти к другим методам отладки.

Проверка кода

В ходе анализа логов вы неизбежно придете в код. На этом этапе можно запустить приложение локально и попытаться воспроизвести ошибку. Учтите, что в процессе отладки вы будете переключаться между различными методами, описанными в этой главе. Поэтому, если логи сразу указывают на проблему в коде, смело применяйте соответствующие рекомендации. Также можно начать с анализа кода, а затем проверить логи. Любой подход, который помогает найти первопричину ошибки, является правильным.

Если вы работаете только с сообщением об ошибке, выполните поиск по проекту. Если сообщение содержится в коде, поиск покажет его местоположение. Затем перейдите в этот файл и начните отслеживать, что вызывает ошибку. Здесь важно фиксировать, какие участки кода вы уже проверили, так как при переключении между файлами легко потеряться. Также может оказаться полезным выводить файлы попарно на экран и сравнивать их.

Использование сообщений console.log

При локальном запуске приложения и анализе кода добавьте вызовы `console.log` в областях, которые могут быть полезны. Это похоже на просмотр логов, о котором говорилось ранее, но выполняется в консоли браузера — такие записи следует удалять после обнаружения ошибки. С помощью `console.log` можно отслеживать рендеринг компонентов и данные, с которыми они отображаются. Эти программные проверки показывают, когда данные приходят в неожиданном формате или отсутствуют вовсе.

Например, если компонент не отображается, `console.log` позволяет в реальном времени проверить соблюдение условий для его рендеринга. С его помощью можно

обнаружить, что ответ застрял в состоянии загрузки, возвращается пустой массив или доступ к значению объекта выполняется некорректно. Иногда удается выявить побочный эффект, когда компонент загружается при первоначальном рендеринге, но хук `useEffect` вызывает некорректный ререндеринг.

Преимущество console.log в том, что сообщения не требуют особого форматирования — их все равно удалят после решения проблемы. Такие инструменты, как missionlog (https://oreil.ly/uq2qv), pino (https://oreil.ly/Vjak7) и winston (https://oreil.ly/1UAdr), также полезны для отладки фронтенда. Вот пример использования `console.log` для отладки компонента:

```
useEffect(() => {
  console.log('userResponseData', userResponseData)
  console.log('orderResponseData', orderResponseData)
  setUserInfo(userResponseData)
  setOrders(orderResponseData)
}, [orderData, userData])

if (userIsLoading)
  return <CircularProgress data-testid="user-loading-circle" />
console.log('Успешно завершено состояние загрузки пользователя')
if (userErrors || ordersErrors)  showBoundary(userErrors || ordersErrors)
console.log('Успешно пройдены обработчики ошибок')
```

Это небольшой пример использования `console.log` для локальной отладки кода. Такие проверки временны и не должны логироваться в продакшене. Этот подход также помогает проверять инкрементальные изменения кода. Во время отладки поэтапное изменение кода (по одной строке за раз) помогает точнее определить необходимые правки.

Например, вам может потребоваться обновлять URL с параметрами строки запроса, которые меняются в зависимости от выбора пользователя в выпадающих списках. Возможно, вы обнаружите, что строка запроса обновляется не так, как ожидалось, но страница все равно перезагружается. В таком случае вы можете изменить одно из значений выпадающего списка, используя переменную состояния, и проверить, решит ли это проблему. Если да, аналогичным образом измените другое значение и убедитесь, что обновление по-прежнему работает. При отладке важно не допустить ситуации, когда баг исправлен, но вы не понимаете, как это произошло. Или хуже: вы исправили исходный баг, но создали новый. Модульные тесты помогают предотвратить это, хотя и не являются панацеей. Поэтому вносите изменения постепенно и отслеживайте их эффект. Если потребуется менять по одной строке кода за раз — это приемлемо.

Использование точек останова

Теперь, когда вы увидели, как можно использовать console.log для пошаговой отладки кода, можно пойти дальше и применить точки останова. При работе с отладчиком, например в Chrome DevTools, *точка останова* — это место, где выполнение

кода приостанавливается. Это позволяет проверять значения в реальном времени без необходимости расставлять `console.log` повсюду. Точки останова также дают возможность увидеть стек вызовов до момента остановки выполнения, что может помочь выявить истинную причину проблемы.

Вы сможете проходить код построчно, пока не найдете ошибку. Таким образом, вы увидите, какие данные передаются, когда возникает ошибка, и в каком порядке вызываются функции. Если вызывается функция, обычно можно войти в нее, чтобы посмотреть, какие значения она получает и что возвращает. Важное замечание: при работе с асинхронными задачами (например, вызовами API) нельзя напрямую войти в функцию — нужно размещать точку останова внутри самой функции обратного вызова.

Такой подход часто оказывается очень эффективным и лаконичным, поскольку позволяет отслеживать поведение нового кода. Например, вы можете изменить параметр функции с массива на объект. Теперь, когда вы понимаете, какое исправление устранило баг, можно почистить связанный с ним код. Затем можно делать инкрементальные коммиты в Git, чтобы не потерять наработки. Что может быть хуже, чем получить новый баг? Исправить баг, но случайно удалить код.

Использование модульных тестов

Поскольку у вас уже есть тесты, о которых мы говорили в главе 21, их можно использовать и для отладки. Сначала запустите модульные тесты и проверьте, проходят ли они. Затем измените условия тестов и посмотрите, по-прежнему ли результат положительный. Если тест проходит, хотя не должен, — это признак, что нужно внимательнее изучить соответствующий участок кода. Возможно, проблема кроется в асинхронной функции или компоненты отображаются не так, как ожидалось.

Это хороший момент для добавления новых тестов. Иногда построчный анализ кода в поиске новых участков для тестирования помогает выявить скрытые условия. Такой анализ можно использовать для рефакторинга — упрощения кода и обнаружения проблем.

Использование Git

Git может стать вашим помощником при отладке. Он позволяет не только отслеживать изменения, вносимые для исправления ошибки, но и быстро найти коммит, в котором эта ошибка появилась. В этом поможет команда `git bisect` (https://oreil.ly/zVafD) — мы подробнее рассмотрим ее в главе 28. Анализ истории Git и просмотр изменений, сделанных до появления ошибки, часто дает возможность сразу выявить проблемный коммит:

- изучить объединенные пул-реквесты;
- сравнить текущий код с последним коммитом в IDE, как показано на рис. 22.1.

Рис. 22.1. История Git в VS Code

Вы увидите, кто и когда вносил изменения. Важно понимать, что цель этого анализа — не найти виноватого, а решить проблему. Каждому из нас доводилось пропускать через ревью пул-реквестов нестабильный код, который затем вызывал проблемы в продакшене. Зная, как это неприятно, поддержите коллег в подобных ситуациях.

После выявления автора проблемного кода можно связаться с ним, чтобы понять, что он пытался реализовать, и вместе придумать более удачное решение. Лучше всего отладка идет во время парного программирования. Когда два человека вместе разбирают проблему, они могут рассуждать о ней вслух, чтобы таким образом выяснить возможные причины и найти оптимальный путь решения. Как правило, если вы отлаживаете код больше 30 минут, стоит поделиться своими соображениями с другим разработчиком.

Даже просто формулирование вопроса иногда выводит из тупика. В таких случаях полезными будут инструменты ИИ, например ChatGPT (https://openai.com/chatgpt). Когда вы объясняете проблему кому-то другому, это часто позволяет *вам* увидеть ее под новым углом. Вы никогда не знаете, какой опыт есть у ваших коллег: возможно, вы столкнулись с проблемой преобразования дат и времени, а кто-то из команды как раз занимался ею на предыдущем месте работы. Многие ошибочно думают, что они должны мгновенно находить решения для любых проблем. Но никто не может знать всего — даже имея богатый опыт. Поэтому, если зашли в тупик, не стесняйтесь обращаться за помощью.

Использование инструментов разработчика в браузере

При поиске причины ошибки используйте инструменты разработчика браузера (DevTools), чтобы наблюдать за происходящим на странице напрямую. В примерах я буду ссылаться на Chrome DevTools, но аналогичные средства есть во всех современных браузерах. Поскольку вы уже знакомились с этими инструментами, когда анализировали производительность и доступность, мы сосредоточимся на четырех вкладках:

- Elements (Элементы);
- Sources (Источники);
- Network (Сеть);
- Application (Приложение).

Вкладка Elements

Если вам нужно исправить проблему со стилями, но вы не можете найти в коде элемент с некорректным оформлением, поможет вкладка Elements. Перейдите на страницу, щелкните правой кнопкой мыши по нужному элементу — откроется интерфейс, показанный на рис. 22.2 и 22.3.

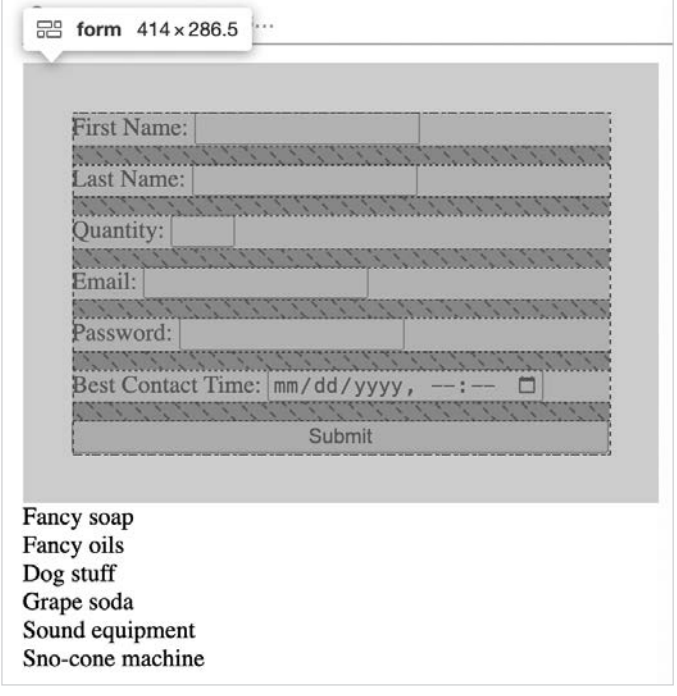

Рис. 22.2. Проверка стилей во вкладке Elements Chrome DevTools

На правой панели (рис. 22.3) отображаются все стили, примененные к выделенному элементу. Вы можете редактировать их прямо в браузере, изменяя значения на этой панели. Это особенно полезно для глубоко вложенных элементов, стили которых задаются родительскими компонентами или сторонними библиотеками (например, MUI), — так вы сможете быстро найти первопричину проблемы и понять, какие стили нужно обновить. Во вкладке Elements также можно проверить состояния при наведении (hover) и мобильные представления.

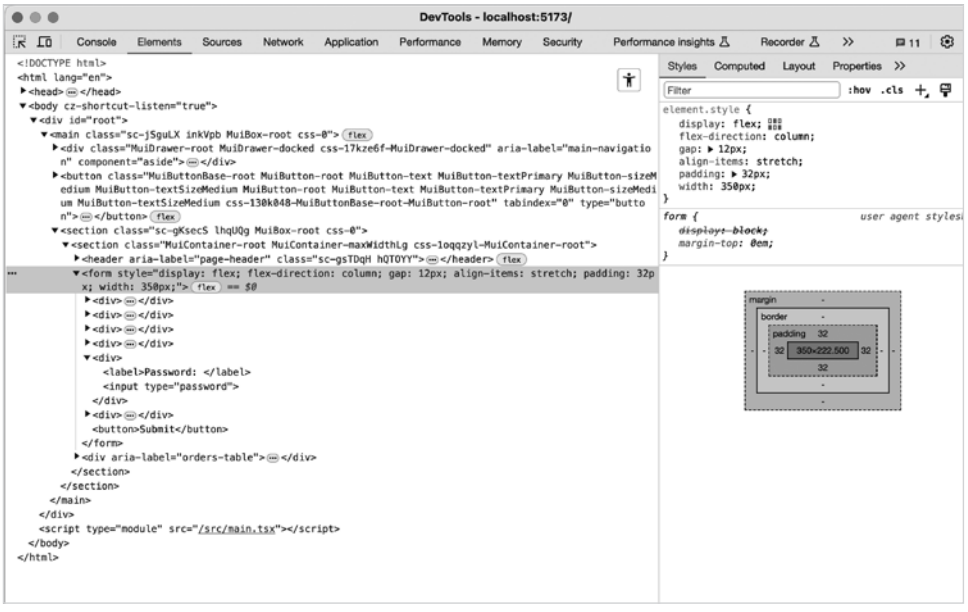

Рис. 22.3. Интерфейс проверки стилей во вкладке Elements Chrome DevTools

Чтобы проверить мобильные представления, нажмите вторую иконку слева в заголовке DevTools (рис. 22.4). Браузер переключит приложение в мобильный или адаптивный режим, который можно настроить с помощью параметров из выпадающего списка (рис. 22.5).

Вкладка Sources

Если вам нужно проверить состояния, рендеринг или моменты, когда устанавливаются значения, вам поможет вкладка Sources. Она позволяет просматривать файлы приложения и устанавливать точки останова для пошагового выполнения кода. Во вкладке Sources можно нажать Cmd-P на Mac или Ctrl+P в Windows, чтобы найти нужный файл (рис. 22.6). Все файлы из вашего репозитория должны быть доступны во вкладке Sources при локальном запуске приложения. Однако в других средах, например в продакшене, точный файл может быть недоступен, так как код был собран и минимизирован.

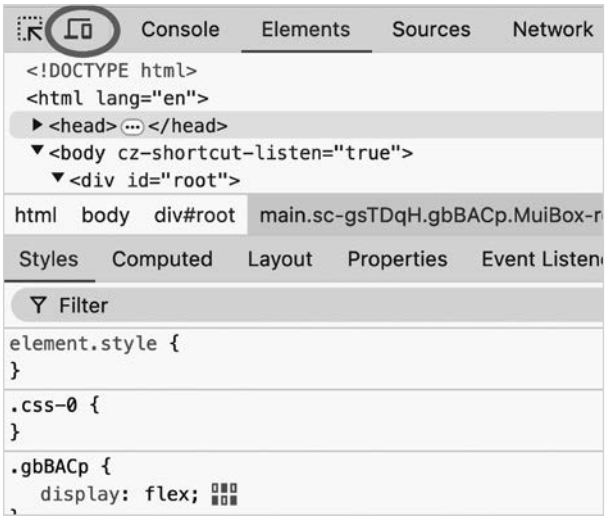

Рис. 22.4. Доступ к мобильному просмотру в Chrome DevTools

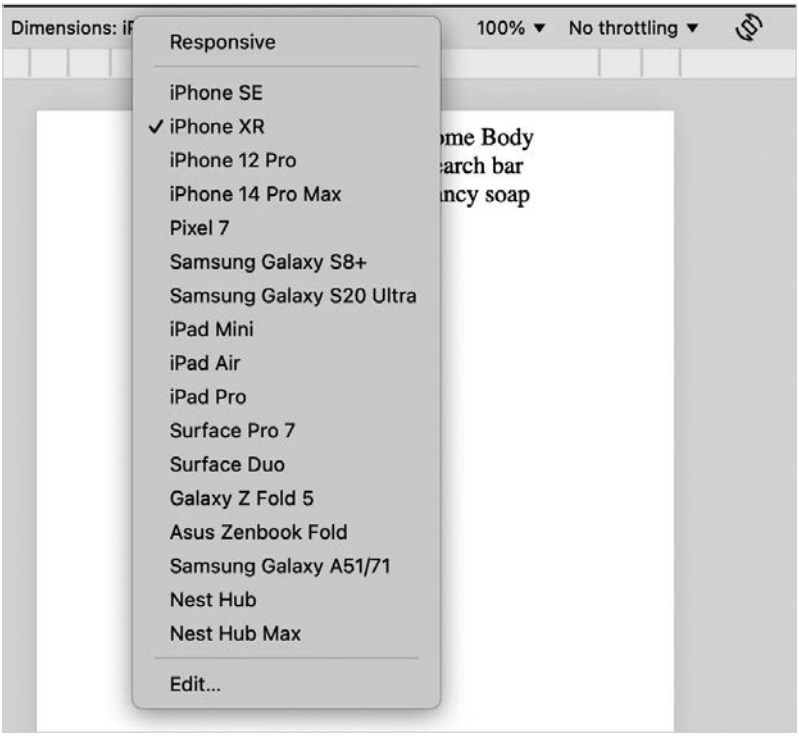

Рис. 22.5. Настройки адаптивности в мобильном просмотре

Рис. 22.6. Поиск файла во вкладке Sources Chrome DevTools

Найдя нужный файл для отладки, вы можете щелкнуть по номеру строки рядом с интересующим кодом, чтобы установить точку останова (рис. 22.7). При обновлении страницы вы увидите, срабатывает ли точка останова. Если она не срабатывает, хотя должна была, это дает новый отправной момент для поиска ошибки. Если же точка останова срабатывает, вы можете просмотреть все данные компонента на этот момент и выполнять код пошагово с помощью панели справа.

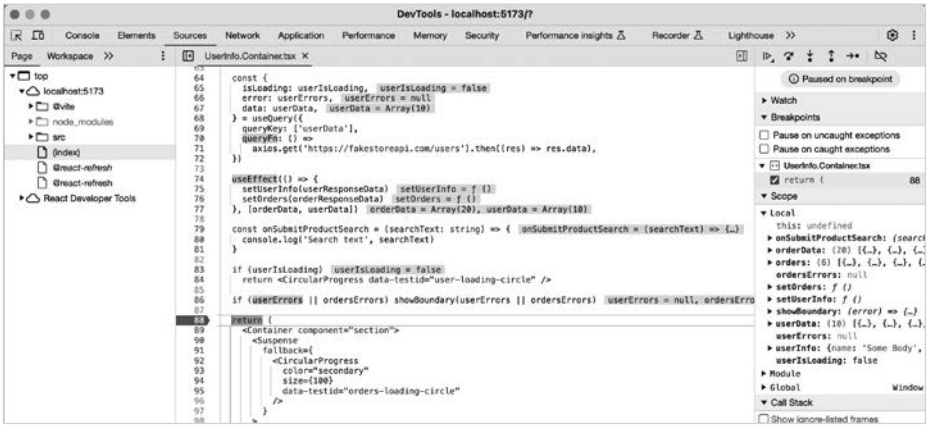

Рис. 22.7. Установка точки останова во вкладке Sources Chrome DevTools

На рис. 22.7 показано, как проверить, не происходит ли что-то неожиданное с переменными состояния, вызовами API или другими функциями. Допустим, вам нужно выяснить причину ошибки, возникающей при попытке пользователя отправить форму. С помощью пошагового выполнения с точками останова вы можете обнаружить, что ввод обрабатывается некорректно. Если точка останова активна и вы обновляете страницу, изначально может отображаться отсутствие данных — в этом случае состояния загрузки предотвратят аварию приложения. Продолжая переходы между точками останова, вы рано или поздно попадете в такое

состояние работы приложения, при котором данные загружаются. В этот момент вы сможете проверить возможные проблемы: например, некорректные данные или ошибочный вызов функции.

Вкладка Network

Если вы определили, что проблема связана с данными, используйте вкладку Network («Сеть») для анализа запросов к бэкенду и получаемых ответов. Здесь отображаются все запросы, которые приложение отправляет при загрузке страницы, включая запросы компонентов, пакетов и вызовы API. Если вас интересует конкретный запрос к API, вы можете проверить переданные заголовки и убедиться, что все параметры указаны верно. Пример запроса к API во вкладке Network показан на рис. 22.8.

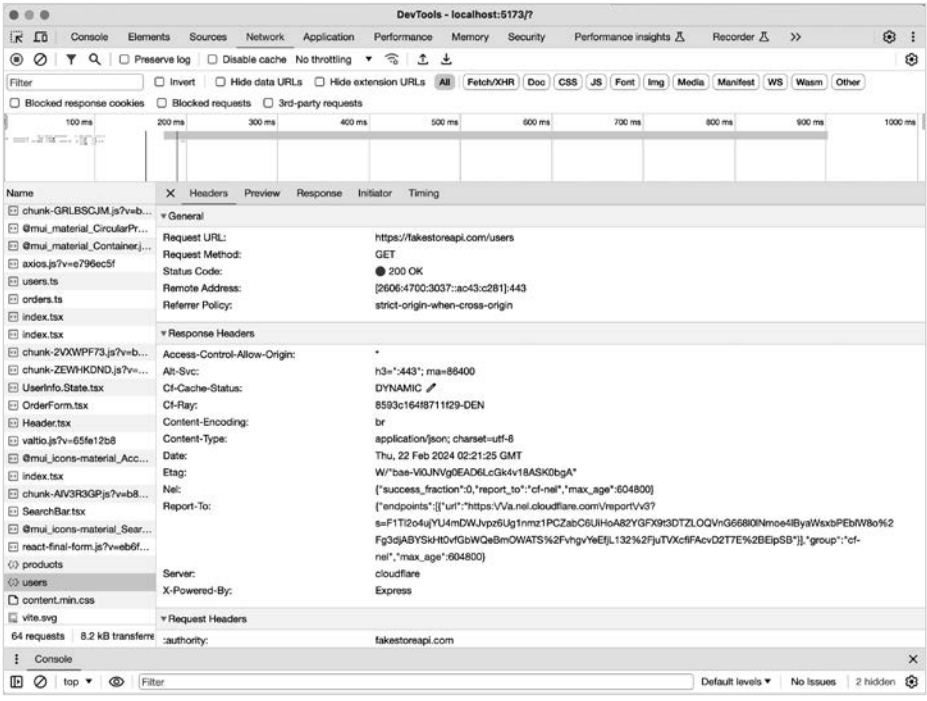

Рис. 22.8. Анализ запроса к API во вкладке Network

Если вы видите ошибку авторизации, например код состояния 403 (https://oreil.ly/IGQmx), проверьте учетные данные в запросе. Возможно, вызываемый URL некорректен: например, должен использоваться «верблюжий регистр» (camel case) или другой формат. Вы также можете проверить, был ли вызван API и обрабатывает ли приложение возвращаемые коды состояния. Вкладка Network отображает весь сетевой трафик страницы.

Инструменты разработчика (DevTools) работают на любом сайте, поэтому я настоятельно рекомендую перейти на часто посещаемый вами ресурс и изучить содержимое вкладки «Сеть» (Network). Скорее всего, вы увидите беспорядочный набор нечитаемых фрагментов и вызовов — и в большинстве случаев именно это вы и должны наблюдать в продакшене. Такое поведение обусловлено соображениями безопасности. Если API и их данные будут доступны всем так же легко, как и вам в локальной среде, это создаст серьезную угрозу безопасности. Именно поэтому я уделяла особое внимание вопросам защиты на протяжении всей книги — чтобы ваше приложение не оказалось в продакшене с полностью прозрачными данными, доступными любому пользователю для просмотра.

Найдя интересующий вас API-запрос, вы можете просмотреть ответ либо в окне Preview, либо в окне Response. Окно Preview отображает ответ в формате JSON, а Response показывает те же данные, но в более развернутом виде. Это позволяет анализировать данные напрямую из API-ответа, не обращаясь к коду приложения. Вы можете проверить, соответствует ли структура данных ожидаемой и были ли изменены поля.

Такие ситуации иногда возникают неожиданно при работе со сторонними сервисами. Они не всегда уведомляют о внесенных изменениях в API, поэтому вы можете узнать о них одновременно с пользователями. На рис. 22.9 показан пример ответа стороннего API.

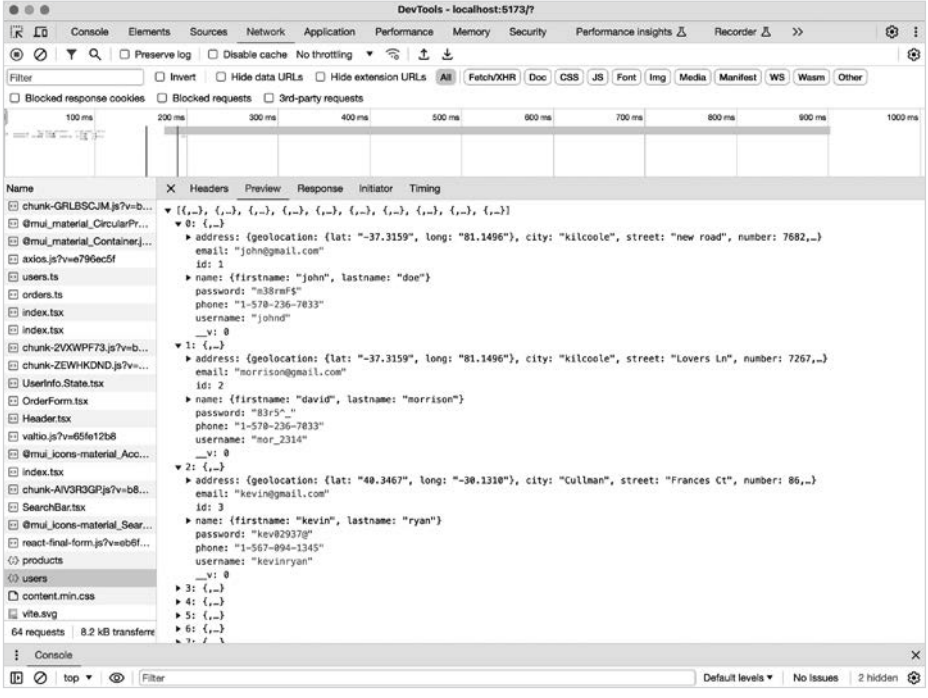

Рис. 22.9. Проверка ответа API во вкладке Network

Во время отладки, если вы заметили, что компонент зависает в состоянии загрузки или постоянно выдает одну и ту же ошибку, попробуйте проверить вкладку Network. Это может указать новое направление для поиска или хотя бы исключить одну из гипотез. Последняя вкладка, которую мы разберем, тесно связана с Network — это вкладка Application («Приложение»).

Вкладка Application

Во вкладке Application отображаются `cookies`, а также значения `localStorage` и `sessionStorage`. В них хранятся пары «ключ–значение», включая:

- токены доступа (access tokens);
- языковые предпочтения пользователя;
- идентификаторы приложения (app IDs);
- другие данные для персонализации приложения.

Если вы сталкиваетесь с постоянными ошибками 403, проверьте эту вкладку — возможно, отсутствует необходимый токен. Для токенов в формате JWT (JSON Web Token) можно использовать декодер вроде jwt.io (https://jwt.io). Иногда проблема заключается просто в недостаточных правах пользователя для доступа к эндпоинту.

Также в хранилище может отсутствовать обязательное значение. Важно помнить: `localStorage` сохраняет данные даже после закрытия вкладки, тогда как `sessionStorage` очищается при ее закрытии или выходе пользователя из системы. Это тоже может быть причиной ошибки. Пользовательские настройки (например, тема интерфейса) обычно хранятся в `localStorage`, а временные состояния (например, флаги показа всплывающих окон) — в `sessionStorage`. Возможность работы с несколькими вкладками одного сайта с разными состояниями обеспечивается именно `sessionStorage` — учитывайте этот факт при отладке.

На рис. 22.10 показано, как может выглядеть `localStorage` в рабочем приложении во вкладке Application.

DevTools, то есть инструменты разработчика в браузере, — это невероятно мощный инструмент отладки, который может рассказать о вашем приложении больше, чем логи или даже сам код. Они помогают увидеть, что происходит во время выполнения, когда код уже собран и минимизирован. Иногда это единственный способ отследить проблему, особенно если она связана со стилями. Как только вы найдете несколько мест, которые могут быть источником ошибки, можно приступать к детальному анализу информации из логов.

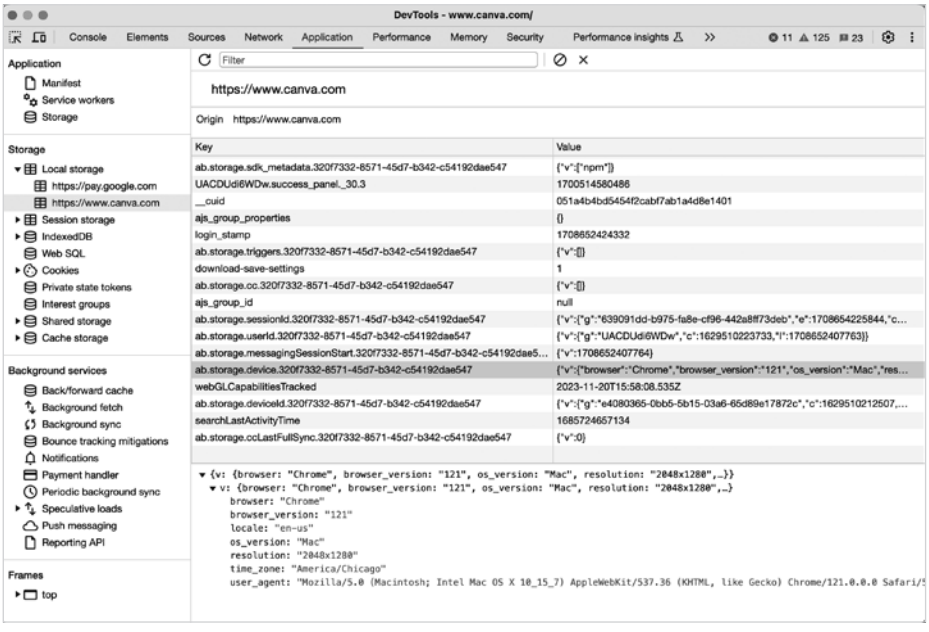

Рис. 22.10. Проверка значений в `localStorage` во вкладке Application

Отладка в других средах

После проверки всех вкладок в DevTools и мобильного представления приложения стоит протестировать его в других средах. Возможно, ошибка не воспроизводится локально — в таком случае проверьте, как ведет себя приложение в среде разработки или в стейджинг-среде. Если ошибка зависит от продакшен-ресурсов (например, подключений к сторонним сервисам), может потребоваться отладка непосредственно в продакшене. При этом обязательно используйте тестовый аккаунт вместо аккаунта реального пользователя, чтобы изменения настроек или значений ни на кого не повлияли.

Можно также привлечь других разработчиков для совместного анализа ошибки в этих общих средах. Стейджинг-среды удобны для вовлечения продуктовой команды — так вы сможете согласовать с ними ожидаемое поведение приложения. Дополнительно попросите членов команды проверить общие компоненты среды в разных браузерах и при различных адаптивных размерах экрана. Лично мне это не раз помогало выявить сложноуловимые баги, когда я слишком зацикливалась на десктопной версии в Chrome. Помните: пользователи взаимодействуют с приложением не так, как вы. Поэтому при обнаружении ошибок старайтесь отлаживать их с точки зрения конечного пользователя.

Отладка в неожиданных местах

Ошибки не всегда возникают из-за кода, данных или даже спецификаций функциональности. Иногда их источником становятся самые неожиданные места, и поиск может занять гораздо больше времени, чем предполагалось. В таких случаях потребуется привлечь специалистов из других команд и выйти за рамки приложения, за которое отвечает ваша команда.

Вам предстоит столкнуться со множеством подобных проблем. Это те самые ошибки, о которых вы будете вспоминать и рассказывать, потому что никто не подозревал об их истинной причине. Вот несколько примеров из моего опыта работы в разных компаниях.

Пользователи всегда находят новые способы взаимодействия с вашими приложениями, которые вы не предусмотрели. Однажды пользователь зашел на сайт со старого планшета, и часть интерфейса не отображалась, потому что наши брейкпоинты адаптивного дизайна не учитывали этот размер экрана. На выявление и исправление проблемы ушло несколько дней. В другом случае пользователь пытался работать с приложением в браузере Safari, и стили страницы отображались некорректно.

Была ситуация, когда пользователи не могли видеть свои данные, потому что обновление стороннего сервиса без предупреждения отозвало разрешения нашего приложения и им пришлось заново предоставлять доступ. В другом случае некоторые пользователи сталкивались с несогласованными обновлениями при заказе товаров, потому что сервер приложения, сервер системы учета запасов и сервер стороннего платежного решения были настроены в разных часовых поясах.

Еще один инцидент: наш CI/CD-пайплайн (CI/CD pipeline) показывал успешное завершение развертываний, но приложение не обновлялось для пользователей. После долгих поисков мы обнаружили, что не обновлялся хеш файла, в котором хранился артефакт сборки (build artifact). Это означало, что в нашем пайплайне что-то незаметно ломалось.

Вы можете столкнуться, например, с файлами, которые не загружаются в одной среде из-за проблем с именами. Однажды продакшен-среда перестала работать, потому что сервер требовал, чтобы файлы начинались с заглавной буквы и содержали подчеркивания, хотя в среде разработки наши файлы прекрасно работали с прежними именами.

Не забывайте очищать кэш и проверять, воспроизводится ли проблема в других браузерах. Также убедитесь, что ваш пользовательский интерфейс не рендерится на стороне сервера. Это усложняет отладку, поскольку компоненты SSR (Server-Side Rendering) не перерисовываются.

Если у вас возникает проблема, когда старая версия приложения работает нормально, а новая — нет, проверьте артефакт сборки. То же самое касается ситуации, когда только что выпущенные изменения не отображаются в продакшене. Хотя

проверка хешей артефактов может быть утомительной, если вы уверены, что с кодом все в порядке, это стоит сделать. Возможные скрытые проблемы включают кэширование артефакта или его загрузку в неправильное место.

Также стоит проверить CI/CD-пайплайн на предмет изменений в скриптах развертывания. Возможные причины:

- установлена новая версия Node.js или среды выполнения;
- обновлены некоторые пакеты в пайплайне.

В главе 27 я подробнее расскажу о скриптах, которые могут быть в вашем CI/CD-пайплайне. Про них часто забывают, пока не возникнут проблемы. Если ошибка связана с инфраструктурой, обратитесь к команде, ответственной за ее поддержку. Обработку критических ошибок мы обсуждали в главе 12 — в таких случаях вы можете воспользоваться плейбуком инцидентов, дополнив его данными фронтенда.

Хотя для примера отладки мы использовали Chrome, обязательно тестируйте приложение в разных браузерах. Из-за различий в поддержке новых CSS-правил и веб-технологий поведение может отличаться. Также важно проверять работу приложения при разных размерах окна, учитывая огромное разнообразие устройств пользователей. Такая комплексная проверка помогает выявлять наиболее сложные ошибки.

Заключение

В этой главе я рассмотрела несколько способов отладки кода. Хочу подчеркнуть: иногда лучшим решением будет просто отвлечься от проблемы. Прогуляйтесь, пообедайте или займитесь чем-то другим. Когда вы слишком долго фокусируетесь на ошибке или фрагменте кода, легко зациклиться на одном аспекте. Не сосчитать, сколько раз после часов или дней борьбы получасовая прогулка помогала мне вернуться и исправить баг максимально быстро и просто.

Если кончились идеи, обращайтесь к коллегам. Часто и ошибка и решение оказываются простыми — сложнее всего найти источник проблемы. Работая в команде, вы получаете доступ к коллективному и разнообразному опыту. Даже если вы не сталкивались с конкретным багом или инструментом, кто-то из команды может обладать нужной экспертизой. Отладка — не всегда индивидуальный процесс, особенно когда нужно срочно исправить баг в продакшене, который мешает пользователям. Не стесняйтесь просить помощи — вы не обязаны разбираться во всем в одиночку.

Часть IV

Развертывание фулстек-приложения

ГЛАВА 23

Настройка развертывания фулстек-приложения

Поздравляю! Вы успешно преодолели первоначальные сложности настройки бэкенда и фронтенда приложения с нуля! Теперь, когда приложение стабилизировалось — с утвержденной архитектурой, сторонними сервисами, соглашениями по коду, выбранными пакетами и всеми остальными компонентами, реализованными в ходе чтения этой книги, — вы и ваша команда наконец готовы быстро и эффективно выпускать новые функции. Все дальнейшие усилия будут направлены на развитие продукта и адаптацию к потребностям пользователей.

Теперь пришло время сосредоточиться на настройке пайплайнов развертывания и обеспечении корректного взаимодействия между базой данных, бэкендом, фронтендом и всеми промежуточными процессами. На этом этапе команда ИБ проведет дополнительные тесты, поскольку полноценное фулстек-приложение готово. Прежде чем перейти к таким темам, как CI/CD-пайплайны и вопросы интеграции, важно сделать шаг назад и проанализировать, насколько эффективно вы и ваша команда проявили себя в роли фулстек-разработчиков.

В этой главе мы обсудим:

- какие еще команды участвуют в процессе развертывания;
- как проверять фронтенд и бэкенд;
- демонстрации и командные ретроспективы.

Хотя вы успешно реализовали проект с нуля, важно провести самоанализ и побудить остальных разработчиков сделать то же самое. После развертывания полноценного приложения у вас наверняка появятся идеи, как можно оптимизировать работу над вашими будущими проектами. Начнем с того, что посмотрим, с какими командами вам предстоит взаимодействовать для выполнения всех развертываний.

Команды, участвующие в развертываниях

Как правило, для создания всей инфраструктуры в облаке вам предстоит сотрудничать минимум с одной дополнительной командой, а иногда с несколькими. Чаще всего это команды инфраструктуры, DevOps и SRE.

Команда инфраструктуры обычно управляет самой облачной платформой и занимается оборудованием, сетями, хранилищами и серверными ресурсами. Эта

команда выбирает оптимальные варианты сервисов на основе обсуждений с вами и командой DevOps. Она обладает большим опытом управления системами на базе Linux и помогает внедрять практики безопасности на этом уровне.

Команда DevOps автоматизирует ручные задачи, например перемещение артефактов или создание скриптов для повторяющихся процессов, и внедряет CI/CD-пайплайны с такими инструментами, как CircleCI (https://circleci.com) или GitHub Actions (https://oreil.ly/sCSGO) (подробнее об этом в главе 27). Кроме того, DevOps способствует взаимодействию между командами разработки и эксплуатации. По вопросам автоматизации инфраструктуры, управления конфигурацией и инструментов для быстрой и надежной поставки ПО обращайтесь к команде DevOps.

В крупных организациях часто есть команда SRE, которая отвечает за надежность, производительность и доступность приложений. SRE работают с:

- инструментами мониторинга;
- реагированием на инциденты;
- оптимизацией производительности.

Когда речь заходит о целевых метриках SLO (метрики уровня обслуживания, service-level objectives), обычно имеются в виду метрики, которые отслеживает и поддерживает именно эта команда.

Все перечисленные команды взаимодействуют с разработчиками, чтобы обеспечить приложениям необходимые ресурсы для стабильного функционирования. Каждая отвечает за свой этап развертывания:

- команда инфраструктуры создает основу в облачной платформе;
- команда DevOps настраивает автоматизацию и пайплайны для доставки приложения в нужные сервисы;
- SRE-команда следит за развернутыми приложениями и управляет изменениями для поддержания uptime (времени безотказной работы), производительности и обработки инцидентов.

Полностью разберитесь в процессах развертывания приложения

Вот совет о развертываниях от Джеффа Грэма для всех разработчиков.

> Я призываю всех инженеров осваивать и контролировать процесс развертывания своих приложений. Хотя команды инфраструктуры или DevOps могут помогать (в зависимости от размера компании), крайне важно полностью понимать каждую деталь. Вы лучше всех знаете свой код, тесты, зависимости и команды сборки, поэтому ваши решения будут оптимальными. Помимо улучшения навыков диагностики проблем, это углубит ваши знания о DevOps, облачных сервисах и других аспектах.

Эти команды помогают вам и разработчикам развертывать приложения, абстрагируясь от базовых систем. Вам предстоит многократно обсуждать с ними выход в продакшен, постепенно открывая доступ для большего числа пользователей.

В небольших организациях все эти задачи (DevOps, инфраструктура и SRE) могут лечь на ваши плечи, и придется самостоятельно во всем разбираться. Это может показаться непосильной работой, но не торопитесь, действуйте постепенно: изучайте документацию и консультируйтесь с коллегами, работающими с этими системами. Сообщите руководителю организации, что это не ваша специализация, — так вы сформируете у него правильные ожидания.

Этапы интеграции фронтенда и бэкенда

На этом этапе вы уже разработали бэкенд, протестировали его и добавили вызовы из фронтенда. Сейчас важно тщательно проверить все компоненты, чтобы обеспечить бесперебойный запуск для всех пользователей. В этот момент инфраструктура проходит настоящее испытание, а вы устраняете оставшиеся недоработки. Часть функциональности уже развернута в стейджинге и, возможно, даже в продакшене, но с ограниченным трафиком.

Даже если сейчас все работает идеально, по мере роста числа пользователей и добавления новых функций могут проявиться неожиданные проблемы. Чтобы сделать процесс интеграции фронтенда и бэкенда максимально гладким, я использую следующие шаги.

Шаги для бэкенда

Некоторые разработчики утверждают, что можно начинать с фронтенда, но я считаю, что гораздо проще сначала подготовить все слои, поддерживающие фронтенд. Вы можете начать с фронтенда, если вам так удобнее, но, скорее всего, в итоге придется делать лишнюю работу. Жестких правил, регламентирующих то, что следует делать в первую очередь, не существует — все зависит от специфики приложения.

Проверьте подключение к базе данных с помощью инструмента вроде pgAdmin (https://www.pgadmin.org). Разработка в локальной среде, среде разработки или стейджинг-среде — это одно, а при переходе в продакшен необходимо убедиться, что у вас корректная строка подключения, учетные данные и схема данных. Убедитесь, что используемый продукт настроен правильно и не возвращает ошибки. Если приложение рассчитано на высокую нагрузку, самое время задуматься о пуле соединений (https://oreil.ly/h--Gq). Возможно, вам потребуется привлечь команду инфраструктуры или DevOps, поскольку эти ресурсы обычно подготавливают именно они.

Заполните базу данных для продакшен-среды тестовыми данными, чтобы убедиться в корректности их размещения. Объем данных может быть небольшим, но он должен быть достаточным для проверки работы всех связей, таблиц, индексов и записи

значений. Поскольку вы уже выполняли это при первоначальной настройке, запустите тот же скрипт, возможно, с небольшими изменениями добавляемых данных.

Выполните запросы к добавленным данным. Протестируйте несколько запросов, чтобы удостовериться, что результаты соответствуют ожиданиям. Также попробуйте обновить данные — это проверит корректность применения ограничений (constraints) и выдачу ошибок при недопустимых типах данных.

Настройте подключение к продакшен-базе данных в вашем бэкенд-приложении. После прямого тестирования через инструмент БД вы знаете, что база стабильна и не должна быть источником проблем. Теперь можно уверенно подключить приложение к БД в продакшене, используя заданные переменные окружения (environment variables).

Перепроверьте и убедитесь, что для переменных окружения в продакшен-среде установлены правильные значения. Это легко упустить из виду, когда ваше внимание поглощено выполнением ручного тестирования.

Совместно с командой инфраструктуры настройте бэкенд-сервер. Такая настройка включает в себя:

- настройку безопасности доступа к бэкенду;
- управление SSL-сертификатами;
- определение мест хранения параметров скриптов;
- выбор регионов развертывания приложения;
- инвалидацию кэша, если он используется.

Также настройте роли и разрешения для доступа к облачным сервисам — они настраиваются отдельно от ролей и разрешений внутри приложения.

Настройте логирование, мониторинг и оповещения для базы данных и API бэкенда. Это необходимо сделать до того, как вы откроете приложение для пользователей, поскольку требуется обеспечить его надлежащую поддержку. Мониторинг работы приложения в продакшене поможет вам и команде находить точки для последующего улучшения. Настройте триггеры оповещений для определенных ошибок в логах или для метрик (например, использования CPU и памяти). Это позволит командам ИБ и другим предупреждать потенциальные атаки или командам SRE и DevOps — масштабировать ресурсы.

Проверьте работу бэкенд-API с помощью инструмента вроде Postman (https://www.postman.com). К этому моменту ваш бэкенд должен быть развернут в продакшен-среде, поэтому убедитесь, что запросы возвращают ожидаемые ответы. Проверьте:

- конфигурацию заголовков;
- права доступа;
- процессы кодирования и декодирования данных.

Эту проверку можно автоматизировать с помощью тестов, чтобы зафиксировать базовый уровень ожидаемого поведения.

Убедитесь, что сторонние сервисы работают корректно. Это критически важный этап, поскольку вы переходите от тестовых учетных данных, которые могут иметь ограниченный доступ, к функциональности сторонних сервисов. Внимательно протестируйте как можно больше сценариев вместе с продуктовой командой, чтобы убедиться в правильности настройки как в приложении, так и в панели управления сервиса. Также повторно проверьте переменные окружения — теперь должны использоваться учетные данные для продакшена.

Не забывайте о фоновых и cron-задачах, так как при работе в продакшене они могут вызывать множество проблем и их не стоит оставлять на потом. Запустите каждую задачу и проверьте, корректно ли она обновляет данные. Некоторые задачи сложно протестировать, если они зависят от внешних сервисов, поэтому создайте тикет, чтобы вернуться к ним, как только в этих сервисах появятся данные или события.

Подготовьте план действий на случай сбоев. Мы детальнее разберем это в главах 25 и 26, но уже сейчас у вас должен быть готов план реагирования на инциденты при неудачных развертываниях. План включает стратегию отката (rollback strategy) и согласование сроков с клиентами и стейкхолдерами. Заранее подготовленный план необходим, потому что во время инцидента все стремятся как можно быстрее потушить «пожар» и систематически действовать сложно.

Документируйте все шаги для базы данных и бэкенда. При будущих развертываниях детальная инструкция поможет команде действовать согласованно. Хотя процесс часто автоматизирован, бывает, что автоматизация ломается. Наличие подробного документа гарантирует резервный вариант для всех команд. Создайте специализированную диаграмму архитектуры бэкенда, чтобы визуализировать связи между приложением и инфраструктурными сервисами. Пример показан на рис. 23.1.

Шаги для фронтенда

С работающим бэкендом в продакшене можно приступать к интеграции фронтенда. Все данные теперь доступны через API, поэтому перейдем к развертыванию фронтенда. Некоторые шаги будут аналогичны выполненным для бэкенда.

Работайте в связке с командой DevOps, чтобы корректно настроить размещение фронтенд-приложения. Настройка размещения может немного отличаться от бэкенда, и команда DevOps может выполнять обе задачи одновременно — все зависит от их предпочтений. Необходимо учитывать те же параметры, что и для бэкенда: переменные окружения, регионы и очистку кэша (cache busting). Отмечу, что обработка кэша может отличаться, так как для производительности и бесперебойной работы фронтенду потребуется CDN.

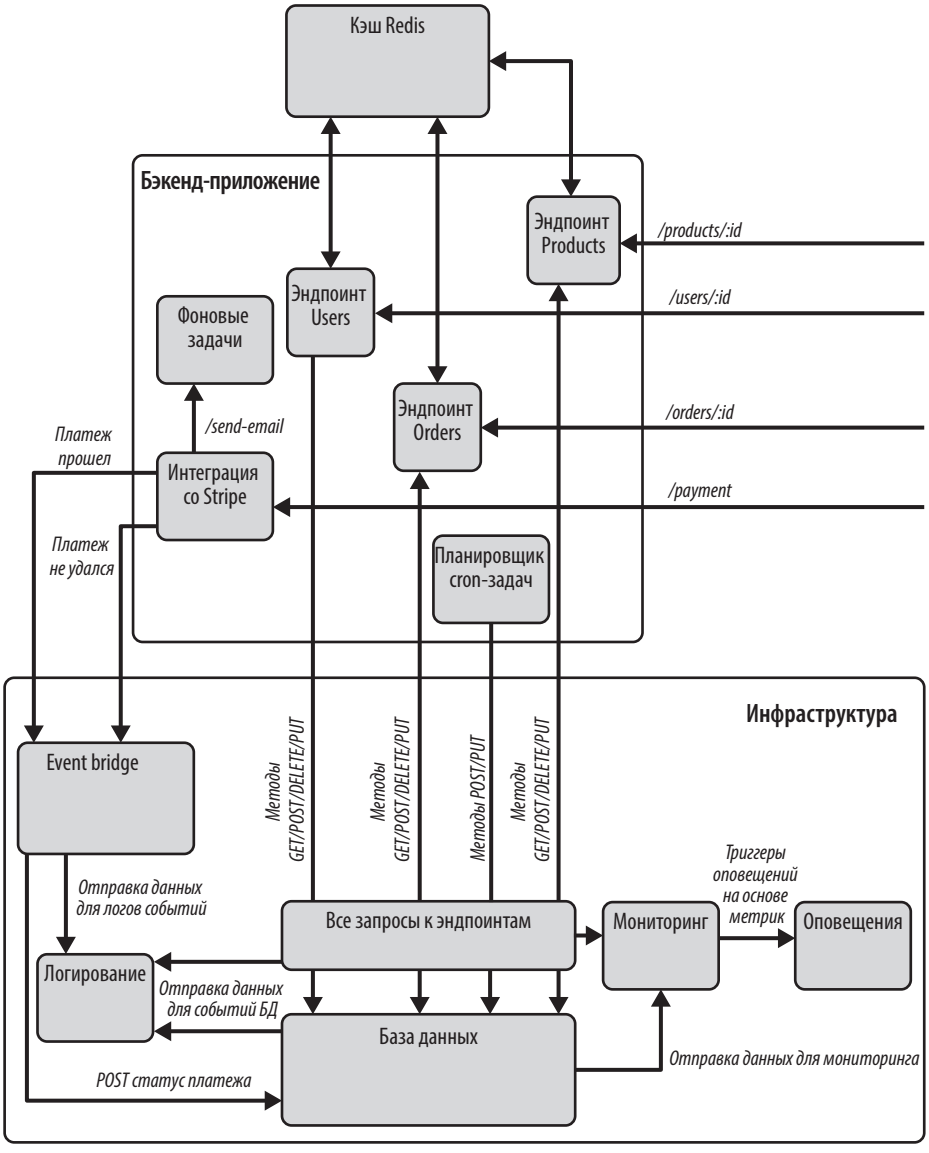

Рис. 23.1. Диаграмма архитектуры бэкенда в продакшене

Внедрите механизмы CDN для ускорения загрузки страниц, как обсуждалось в главе 20. Поскольку вы уже взаимодействуете с командой DevOps, разумно настроить это сразу. Популярный сервис Cloudflare (https://oreil.ly/Thqer) обеспечивает безопасное управление фронтенд-приложениями и содержит встроенные функции для оперативного решения проблем. Альтернативный сервис для приложений, размещенных на AWS, — CloudFront (https://oreil.ly/gxU9o).

Оптимизируйте бандл: убедитесь, что код минимизирован, а ресурсы сжаты. Обычно это происходит в CI/CD-пайплайне, который мы настроим в главе 27. Это артефакт, который будет размещен на фронтенд-сервере и с которым будут взаимодействовать пользователи. Именно поэтому важно минимизировать размер бандла.

Проверьте работу логирования, мониторинга и оповещений. Для многих пользователей фронтенд — это и есть продукт, поэтому за его состоянием нужно следить так же внимательно, как и за бэкендом. Возможно, вам будет удобно использовать отдельные дашборды для логирования и мониторинга фронтенда — это ускорит поиск нужной информации. То же самое касается оповещений: разные специалисты должны получать уведомления, чтобы проблемы решались максимально эффективно.

Убедитесь, что фронтенд работает во всех браузерах, которые вы планируете поддерживать. Часто мы фокусируемся только на браузере, в котором ведем разработку, и не проверяем остальные. Сервис CanIUse (https://oreil.ly/HCY93) дает перечень наиболее популярных браузеров и поддерживаемых ими функций — проверьте работу приложения в этих браузерах после его развертывания. Также используйте такие инструменты, как SauceLabs (https://saucelabs.com), чтобы убедиться, что приложение корректно работает на разных устройствах и браузерах. Это лучше делать еще в процессе тестирования и отладки (как обсуждалось в главе 22), но если вы забыли, то обязательно сделайте это сейчас.

Убедитесь, что все формы работают как ожидается. Вы наверняка столкнетесь с пограничными случаями в будущем, но важно убедиться, что основная функциональность поддерживается корректно. Пользовательский интерфейс должен обновляться соответствующим образом, а для анализа запросов и ответов можно использовать вкладки Network и Application.

Протестируйте приложение через фронтенд и отслеживайте изменения в базе данных в продакшене. Пройдите несколько пользовательских сценариев (user flows) вместе с продуктовой командой, чтобы убедиться, что все работает. Это ключевой момент для проведения демонстраций: проведите совместный звонок, где вы пройдете по приложению, а затем передайте управление продуктовой команде, пусть они тоже потестируют сценарии. Ваше приложение должно быть удобным для тех, кто не участвовал в разработке: мы, разработчики, иногда настолько привыкаем к «костылям», что забываем — никто их в приложении использовать не будет.

Подготовьте документацию для фронтенда. Вы можете детализировать структуру приложения и логику работы компонентов или ограничиться диаграммой высокого уровня — ориентируйтесь на то, как будет удобнее визуализировать интеграцию. Создание диаграммы дерева компонентов (component tree) должно было быть выполнено ранее, но если нет — самое время помочь коллеге. На текущем этапе лучше сохранять высокоуровневый подход, чтобы избежать смешения инфраструктурных деталей с состоянием компонентов. На рис. 23.2 показан пример диаграммы высокого уровня.

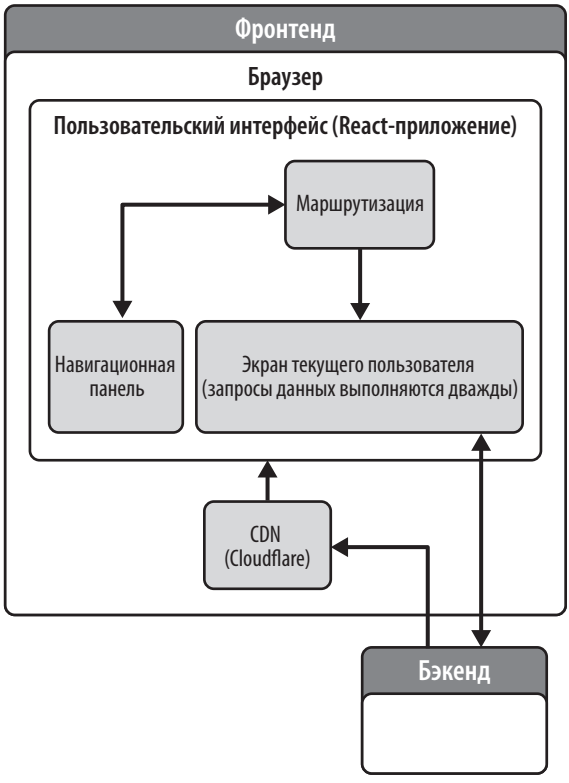

Рис. 23.2. Диаграмма архитектуры фронтенда в продакшене

Завершающие шаги

После развертывания всего стека осталось проверить несколько моментов, чтобы убедиться, что учтено все, что можно. Когда все развернуто, сделайте небольшой перерыв. Получасовая прогулка или просто время вдали от компьютера помогут вернуться к работе со свежей головой.

Теперь, когда приложение полностью в продакшене, важно получить подтверждение того, что вы сможете его поддерживать. Один из способов — проверить логи и удостовериться, что там отображаются тесты подключения. Надеюсь, ошибок пока нет, но активность должна быть видна. Проверьте, корректно ли работает мониторинг: ищите трафик от тестов подключения. Запустите оповещения, чтобы убедиться, что они приходят на нужные email-адреса и каналы. Также протестируйте планы реагирования на инциденты и откаты, чтобы увидеть, как они работают на практике.

Вы не можете быть уверены, что ваш план отката работает, пока не протестируете его, но такой тест может нарушить работу системы. У тех, кто занимается аварийным восстановлением, есть поговорка: «Считайте, что у вас нет плана аварийного

восстановления, если вы ничего не восстанавливали после аварии». Другими словами, пока вы не проверите свой план в действии, вы не можете быть в нем уверены. На самом деле даже реагирование на инциденты и выполнение откатов могут нарушить работу продакшена.

Вам также будет полезно знакомство с используемыми сервисами. Разумеется, можно легко переложить все на команду DevOps, чтобы не погружаться в детали их работы, — в таких случаях достаточно освоить базовые операции. Но в любом случае хорошо бы знать, где в интерфейсе облачного провайдера можно просматривать конфигурации, и хорошо бы изучить документацию провайдера, чтобы получить общее представление о настройках.

Завершающие шаги из этого раздела помогут вам определить основные точки приложения усилий и источники потенциальных проблем. Поскольку каждый проект уникален, дополните этот список шагами, исходя из своего опыта. Многие разработчики за годы работы создают собственные чек-листы, которые затем используют в новых проектах или на новой работе.

Документирование и шаги по поддержке

На этом этапе следует обновить архитектурную диаграмму до актуального состояния продакшена, так как теперь вы понимаете, как все компоненты взаимодействуют между собой. На рис. 23.3 показан пример такой диаграммы для этого приложения.

Теперь, когда у вас есть диаграмма для продакшена, обновляйте ее по мере внесения изменений. Вы можете добавлять в диаграмму столько деталей, сколько посчитаете нужным: например, можно провести линии ко всем доступным эндпоинтам или включить схему базы данных. После выполнения этих шагов и создания собственного чек-листа задокументируйте его, покажите команде и запишите недочеты, которые найдут участники команды, — эти недочеты следует устранить.

Помните: хотя ваше приложение может находиться в продакшен-среде, по-настоящему публичным оно становится только тогда, когда появляются пользователи, работающие с ним. До этого продакшен может не сильно отличаться от других окружений. Несколько раз прогоните процесс выпуска приложения, чтобы каждый раз изменения команды переносились из других окружений в продакшен, и задействуйте каждого разработчика в команде, чтобы он дал обратную связь и подтвердил, что процесс ему удобен. К моменту появления пользователей развертывание в продакшен-среде должно стать рутинной операцией. Также стоит протестировать планы реагирования на инциденты при минимальном трафике, чтобы убедиться в их работоспособности.

Хотя вы прошли этот процесс с нуля, с каждой итерацией вы будете сталкиваться с новыми ошибками, отклонениями или аномалиями. Этот процесс обучения непрерывен, и вас еще ждет много открытий.

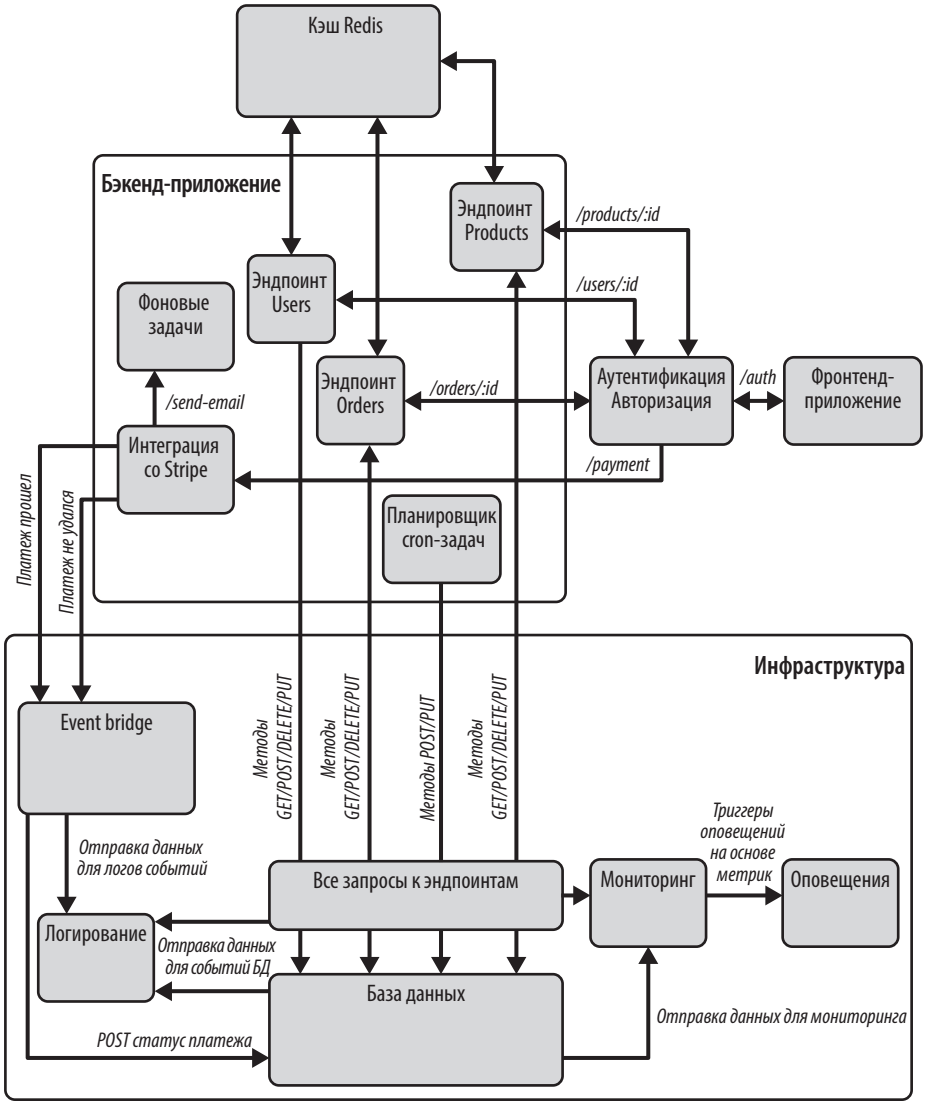

Рис. 23.3. Архитектурная диаграмма продакшена

Заключение

В этой главе мы разобрали основные шаги, которые нужно выполнить, чтобы подготовить приложение к продакшену. После написания и тестирования кода в предварительных средах остается множество нюансов, которые нужно учесть. Независимо от того, выходит ваше фулстек-приложение в продакшен в первый или

в сотый релиз, проблемы возможны. Обычно процесс становится более гладким после нескольких успешных развертываний, но сбои все равно случаются. Перед тем как объявить о готовности изменений, обязательно протестируйте их и будьте готовы выполнить откат или устранить инциденты.

В любом случае вы теперь знакомы с различными аспектами процесса, понимаете, как они взаимодействуют между собой и что за ними стоит. К этому моменту вы уже скоординировали работу нескольких команд, включая команду разработчиков. Теперь можно отточить процесс, задокументировать его и помочь другим членам команды освоить развертывания в продакшен. Чем увереннее команда будет чувствовать себя при таких развертываниях, тем стабильнее они в дальнейшем будут проходить.

ГЛАВА 24

Интеграционное тестирование

Один из лучших способов обеспечить удобство обслуживания ваших приложений в долгосрочной перспективе — писать интеграционные тесты. Когда я описывала тестирование бэкенда и фронтенда в главах 7 и 21, я упоминала, что мы подробнее рассмотрим эти тесты. Сквозные тесты (end-to-end, e2e) — отличный способ убедиться, что ваши изменения не нарушают функциональность всего стека приложения. Эти тесты весьма полезны, когда вы обновляете пакеты или вносите изменения в глобальные компоненты, так как они позволяют выявить непредвиденные побочные эффекты. Они экономят время команды QA на регрессионное тестирование и предотвращают появление багов в неожиданных местах при использовании приложения.

Эти тесты также помогают документировать, как должны работать функции по мере роста приложения. Если вы что-то меняете, например логику клика по кнопке, такие тесты будут полезнее модульных. Сквозные тесты могут имитировать действия пользователя, обеспечивая стабильное воспроизведение реального потока работы. В этом ключевое отличие между модульными и сквозными тестами, и именно поэтому последние дают больше уверенности в том, что функциональность не затронута.

Инструменты для сквозного тестирования, которые мы рассмотрим в этой главе, обычно не зависят от стека технологий, поэтому их можно использовать не только в React-проектах. Я покажу, как писать одни и те же тесты с помощью трех библиотек, чтобы вы могли сравнить их работу:

- Cypress;
- Nightwatch;
- Playwright.

На написание сквозных тестов обычно уходит больше времени, чем на написание модульных, поэтому, несмотря на то что они более тщательно проверяют сценарии взаимодействия, их разработка и поддержка обходятся дороже. Чтобы тест-кейсы оставались управляемыми, полезно определять их во время разработки функций или планирования развития продукта.

Тест-кейсы

Прежде чем перейти к коду, давайте определим три тест-кейса, которые мы напишем для каждого из инструментов.

Первый тест-кейс — проверка загрузки таблицы заказов. Для этой функциональности потребуется несколько ответов от разных эндпоинтов, ожидание изменения состояния загрузки, а также загрузки данных в компоненте таблицы. Поскольку в этом функционале много подвижных частей, продуктовая команда, скорее всего, поможет вам в описании сценариев. Возможно, они будут использовать Gherkin (https://oreil.ly/olEOS), как мы обсуждали в главе 21.

Второй тест-кейс — проверка корректности отправки заказа. В этом случае нужно проверить, что входные значения валидны, эндпоинт вызывается правильно, и после получения ответа от эндпоинта возвращается ожидаемый статус-код. Это позволит убедиться, что пользователи могут успешно оформлять заказы.

Третий тест-кейс — проверка невозможности отправки неполного заказа. Здесь мы проверяем, что при невалидных входных значениях на странице отображается корректное сообщение об ошибке. Этот сценарий подтвердит, что пользователи получают релевантную и полезную обратную связь при вводе некорректных данных.

Теперь, когда тест-кейсы определены, давайте напишем тесты для них с помощью различных инструментов сквозного тестирования.

Сквозные тесты с Cypress

Первым инструментом для сквозного тестирования, который мы рассмотрим, будет Cypress (https://oreil.ly/vLJ1h). Это один из самых популярных инструментов, использующих приложение в реальном браузере, как это делал бы пользователь. Работа с Cypress выглядит так, как будто вы даете человеку инструкции по работе с приложением. Cypress предоставляет методы для взаимодействия с компонентами на странице и вызова реальных API. Вы также можете имитировать вызовы API, если у вас есть тестовые данные, которые будут возвращаться для обеспечения стабильности тестов. Однако помните, что это снижает ценность именно сквозного тестирования. Для начала установите Cypress, если вы еще не сделали этого, изучая главу 21:

```
npm install cypress --save-dev
```

Также потребуется добавить конфигурации в директорию cypress, обновив файл `tsconfig.json`. Обратитесь к документации Cypress (https://oreil.ly/wyNnf), чтобы определить необходимые настройки, так как они часто меняются.

У вас уже есть тесты в `cypress/e2e/user-info.cy.ts`, и мы их расширим. Сначала проведите рефакторинг тестов с помощью хука `beforeEach` — это позволит автоматически выполнять повторяющийся код для каждого теста вместо его

дублирования. Будут выполняться три вызова API: получение заказов, получение данных пользователя и создание заказа. Эти вызовы можно имитировать и перехватывать в каждом тесте. Вот как это реализовать:

```
// user-info.cy.ts
describe('Информация о пользователе', () => {
  beforeEach(function () {
    cy.intercept('GET', '/v1/orders').as('getOrders');
    cy.intercept('GET', '/v1/users').as('getUser');
    cy.intercept('POST', '/v1/orders').as('createOrder');
  });
…
```

Далее добавьте первый новый тест в файл `user-info.cy.ts` под тестом, где вы кликаете на ссылку Actions. Этот тест проверит корректность загрузки таблицы заказов, вызов правильного эндпоинта и завершение загрузки данных. Помните: мой подход — лишь один из множества возможных вариантов написания таких тестов:

```
it('загружает таблицу заказов', function () {
  cy.visit('http://localhost:8080/');
  cy.wait('@getOrders');

  cy.get('[aria-label="orders-table"]')
    .contains('Товары для собак')
    .should('be.visible');

  cy.get(`[data-testid="orders-loading-circle"]`)
    .should('not.be.visible');
});
```

Здесь используется перехват `getOrders`, определенный в хуке `beforeEach`. Вы находите таблицу заказов по атрибуту `aria-label`, проверяете наличие ожидаемого наименования товара (например, «Товары для собак»), чтобы убедиться в корректности данных из ответа. Затем смотрите, чтобы индикатор загрузки таблицы не отображался на странице. Напомню: ранее мы добавили атрибут `data-testid` (https://oreil.ly/tBdEe) для тестирования.

Теперь запустите тест и удостоверьтесь, что он успешно пройден, а результаты корректны. Если тест проходит, намеренно измените одно из утверждений, например, проверку скрытости индикатора загрузки замените на ожидание его видимости. Это спровоцирует падение теста, что подтвердит его корректность и исключит ложноположительный результат.

Следующий тест немного сложнее, поскольку предполагает программное заполнение формы. Необходимо найти все поля ввода, ввести в них корректные значения, затем найти кнопку Submit и нажать ее. После этого можно проверить:

- успешность выполнения API-запросов с помощью ранее определенного мока `createOrder`;
- отображение на странице сообщения об успешной отправке.

Вот как можно написать этот тест:

```
it('отправляет успешный запрос на заказ', function () {
  cy.visit('http://localhost:8080/')
  cy.wait('@getOrders')

  cy.get('input[name="firstName"]').type('Ernest')
  cy.get('input[name="lastName"]').type('Abcde')
  cy.get('input[name="quantity"]').type('3')
  cy.get('input[name="email"]').type('e.abcde@ern.com')
  cy.get('input[name="password"]').type('B1gt3sTAcc0un!')
  cy.get('input[name="contactTime"]').type('2024-07-09T12:00')

  cy.get('form').find('Submit').click()

  cy.wait('@createOrder').its('response.statusCode').should('equal', 204)

  cy.find('Order submitted successfully').should('be.visible')
})
```

Последний тест проверяет корректность валидации формы и ее способность блокировать отправку некорректных данных. В этом тесте заполняются все поля формы, но email указывается в неверном формате, что должно привести к отображению ошибки на странице:

```
it('исключает отправку неполного запроса на заказ', function () {
  cy.visit('http://localhost:8080/');
  cy.wait('@getOrders');

  cy.get('input[name="firstName"]').type('Ernest');
  cy.get('input[name="lastName"]').type('Abcde');
  cy.get('input[name="quantity"]').type('3');
  cy.get('input[name="email"]').type('e.abcde');
  cy.get('input[name="password"]').type('B1gt3sTAcc0un!');
  cy.get('input[name="contactTime"]').type('2024-07-09T12:00');

  cy.get('form').find('Submit').click();

  cy.get('.email-errors').should('be.visible');
});
```

Теперь у вас есть базовый набор тестов, который можно расширять. Запустите эти тесты, чтобы проверить их выполнение. Если тесты не проходят, проанализируйте код компонента, тестовый код и документацию Cypress, чтобы найти причину. Также рекомендуется запустить эти тесты в CI/CD-пайплайне, чтобы оценить время их выполнения.

Сквозные тесты с Playwright

Следующий инструмент сквозного тестирования, который мы реализуем, — Playwright (https://oreil.ly/9VJ0j). Это еще один популярный тестовый фреймворк.

В некоторой степени он похож на Testing Library (https://oreil.ly/yDJWm); Playwright даже предоставляет руководство по миграции (https://oreil.ly/fUDub), где показано, как реализовать аналогичную функциональность. Он автоматически запускает тесты в Chromium, Firefox и WebKit, охватывая три браузера. Конфигурация немного отличается от Cypress. Все настраивается при установке и инициализации пакета следующей командой:

```
npm init playwright@latest
```

При установке будут появляться интерактивные подсказки с перечнем опций для настройки соответствующих конфигураций (https://oreil.ly/Yk4-8). Для этого примера подойдут значения по умолчанию, но можно поэкспериментировать с другими параметрами, чтобы увидеть, что они меняют. После первоначальной настройки переименуйте файл теста Playwright в `user-info.spec.ts`. Затем добавьте следующий код для начала работы:

```
import { test, expect } from '@playwright/test'
import { orderResponseData } from '../src/mocks/orders'

let apiContext

test.beforeEach(async ({ page, playwright }) => {
  await page.goto('http://localhost:8080/')

  apiContext = await playwright.request.newContext({
    baseURL: 'https://api.teststore.com',
    extraHTTPHeaders: {
      Authorization: `token ${process.env.API_TOKEN}`
    }
  })
})

test.afterEach(async () => {
  await apiContext.dispose()
})
```

Аналогично тому, что вы делали в тестах Cypress, здесь используются хуки `beforeEach` и `afterEach` для перехода на локальный URL и настройки мока API. В Playwright вы используете встроенный контекст для тестирования API (https://oreil.ly/VMyJR). Вы можете добавить специфичные заголовки и расширить этот объект для таких задач, как авторизация и валидация токенов. URL https://api.teststore.com — это домен бэкенда, но вы можете заменить его на свой, особенно если запускаете сквозные тесты локально.

Для полноты мы добавим один тест-кейс из главы 21, который ранее написали для Cypress. Этот кейс проверяет базовую навигацию и определяет, работает ли ссылка:

```
test.describe('User Info', () => {
  test('переходит на страницу действий TestStore', async ({ page }) => {
    const actionsButton = page.getByText('Действия')
```

```
    await actionsButton.click()

    expect(page.url().includes('/actions'))
  })
})
```

Затем вы добавите в этот файл те же три тест-кейса, которые писали для тестов Cypress:

```
test('загружает таблицу заказов', async ({ page }) => {
  const ordersData = await apiContext.get(`/v1/orders`, {
    data: {
      orderResponseData,
    },
  })

  expect(ordersData.ok()).toBeTruthy()

  const orderRow = page.getByText('Товары для собак')

  expect(orderRow).toBeVisible()
})
```

Здесь вы настраиваете контекст на возврат имитационного ответа от эндпоинта заказов и проверяете, что страница загружается с этими данными. Далее приведен пример выполнения POST-запроса при отправке формы:

```
test('отправляет успешный запрос на заказ', async ({ page }) => {
  page.getByLabel('Имя').fill('Ernest')
  page.getByLabel('Фамилия').fill('Abcde')
  page.getByLabel('Количество').fill('3')
  page.getByLabel('Email').fill('e.abcde@ern.com')
  page.getByLabel('Пароль').fill('B1gt3sTAcc0un!')
  page.getByLabel('Время контакта').fill('2024-07-09T12:00')

  const submitButton = page.getByText('Отправить')

  await submitButton.click()

  const ordersData = await apiContext.post(`/v1/orders`, {
    data: {
      firstName: 'Ernest',
      lastName: 'Abcde',
      quantity: 3,
      email: 'e.abcde@ern.com',
      password: 'B1gt3sTAcc0un!',
      contactTime: '2024-07-09T12:00',
    },
  })

  expect(ordersData.ok()).toBeTruthy()
```

```
  expect(page.getByText('Заказ успешно отправлен')).toBeVisible()
})
```

Заполнение полей формы аналогично тому, что вы делали в Cypress: вы находите поле и вводите нужное значение. Однако способ отправки POST-запроса и проверки его успешности здесь немного отличается от Cypress.

Последний тест-кейс проверяет ошибку валидации поля email:

```
test('не отправляет неполный запрос на заказ', async ({ page }) => {
  page.getByLabel('Имя').fill('Ernest')
  page.getByLabel('Фамилия').fill('Abcde')
  page.getByLabel('Количество').fill('3')
  page.getByLabel('Email').fill('e.abcde')
  page.getByLabel('Пароль').fill('B1gt3sTAcc0un!')
  page.getByLabel('Время контакта').fill('2024-07-09T12:00')

  const submitButton = page.getByText('Отправить')

  await submitButton.click()

  const formError = page.getByText(
    "Пожалуйста, укажите '@' в адресе email."
  )

  expect(formError).toBeTruthy()
})
```

Это та же самая проверка, что и в тесте Cypress, но с другим синтаксисом. Теперь вы можете запустить тесты следующей командой и проверить их результаты:

```
npx playwright test
```

На рис. 24.1 показан пример скриншота с результатами тестов в браузере. Playwright генерирует файл `index.html`, который можно:

- добавить в репозиторий;
- исключить из репозитория, но при этом отображать в CI/CD-пайплайне.

В результатах тестов отображается, какие тесты прошли успешно, а какие завершились с ошибкой, а также в каких браузерах они выполнялись. Если кликнуть на результат теста, откроется детализированное представление с информацией о причинах успешного или неуспешного прохождения теста. На рис. 24.2 показан пример детализации проваленного теста.

Теперь можно перейти к проблемному участку теста и начать отладку. В Playwright есть удобные инструменты отладки, описанные в документации (https://oreil.ly/e8kyy), например:

- точки останова;
- подробные логи API.

Точки останова работают аналогично тем, что вы видите в браузере.

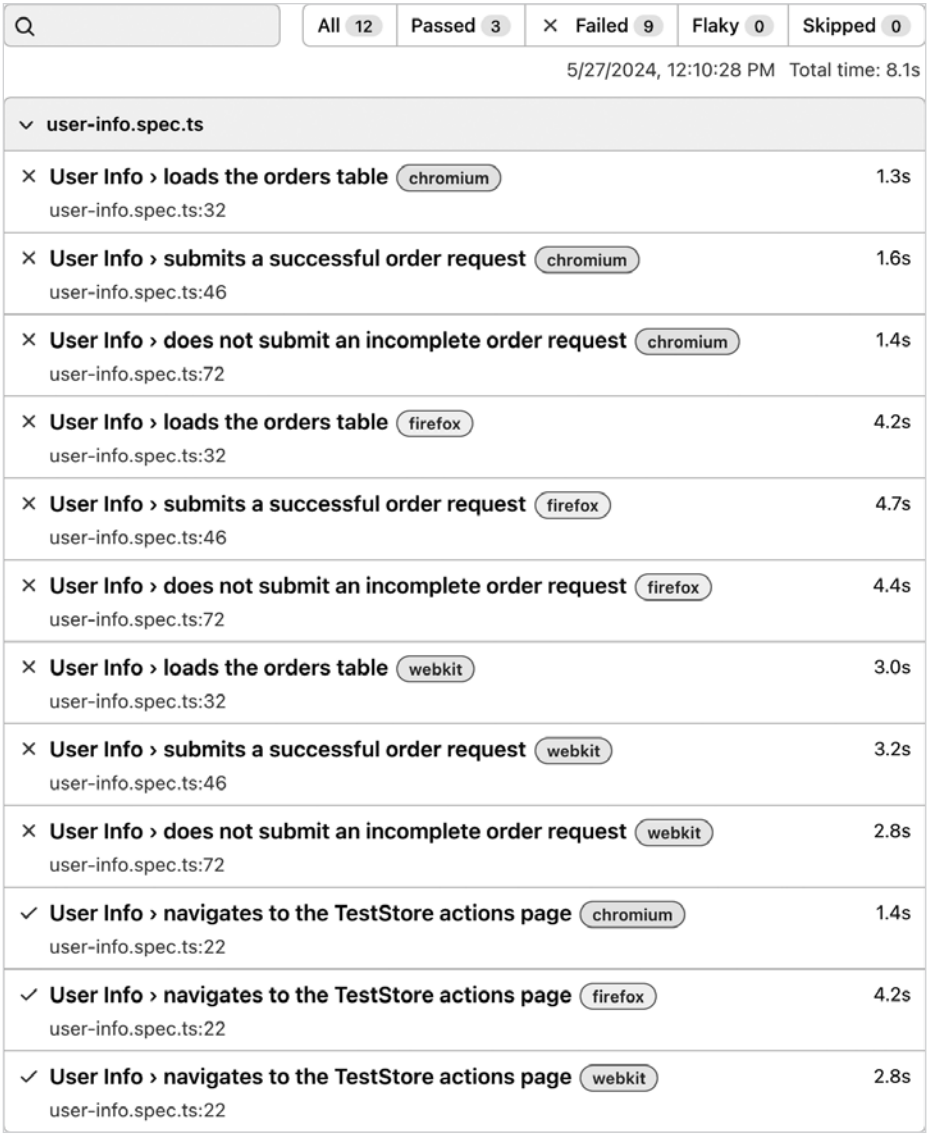

Рис. 24.1. Обзор результатов тестирования в Playwright

Рис. 24.2. Детализированное представление проваленного теста

Сквозное тестирование с помощью Nightwatch

Последний инструмент, который мы реализуем, — Nightwatch (https://oreil.ly/vqctm). Он основан на Selenium WebDriver (https://oreil.ly/Ky66A) — одном из старейших инструментов для автоматизации браузеров. Многие инструменты тестирования используют Selenium в проектах различных типов. Также есть возможность запускать тесты на удаленном сервере Selenium, аналогично Cypress Cloud (https://www.cypress.io/cloud). Поэтому, если вы или кто-то из вашей команды знаком с Selenium, это хороший вариант, так как вы сможете использовать имеющиеся навыки сотрудника. Еще одно преимущество Nightwatch — интеграция с SauceLabs (https://saucelabs.com) и другими кросс-платформенными инструментами, если вам нужно тестировать на широком спектре устройств.

По этой команде устанавливается Nightwatch, он предложит ответить на вопросы настройки:

```
npm init nightwatch
```

Можно принять значения по умолчанию для вопросов настройки в терминале. Снова рекомендую уделить время изучению параметров конфигурации (https://oreil.ly/MW3hd) в документации Nightwatch.

После завершения начальной настройки необходимо установить следующий пакет Nightwatch, чтобы тестировать API-запросы, как и в других инструментах сквозного тестирования:

```
npm i @nightwatch/apitesting --save-dev
```

В отличие от других инструментов, Nightwatch использует отдельный пакет, чтобы сохранить фокус на одной задаче, поскольку часто сквозные тесты должны обращаться к реальным эндпоинтам. Однако в нашем случае мы используем мок-данные.

Теперь нужно обновить файл `nightwatch.conf.cjs`, добавив новый плагин и объект `api_testing`, как описано в документации Nightwatch (https://oreil.ly/rN84U). Это завершающий этап настройки перед тестированием. Вы создадите те же четыре тест-кейса, что и в предыдущих примерах с Cypress и Playwright, выполнив начальную настройку для всех сценариев. Начните с настройки мок-сервера API, запросов и данных:

```
import { ExtendDescribeThis } from 'nightwatch';
import { orderResponseData } from '../src/mocks/orders';

interface CustomThis {
  customerPortalUrl: string;
  submitButton: string;
}
```

```
describe('Информация о пользователе', function (this:
ExtendDescribeThis<CustomThis>) {
  this.customerPortalUrl = 'http://localhost:8080/';
  this.submitButton = '*[type=submit]';

  let server;

  beforeEach(async function (
    this: ExtendDescribeThis<CustomThis>,
    browser,
    client
  ) {
    server = await client.mockserver.create();

    server.setup((app) => {
      app.get('/v1/orders/', function (_, res) {
        res.status(204).data(orderResponseData);
      }),
      app.post('/v1/orders/', function (_, res) {
        res.status(204).data([]);
      });
    });

    await server.start(3000);

    browser.navigateTo(this.customerPortalUrl!);
  });

  afterEach(() => {
    server.close();
  });
```

Этот код выполняет те же функции, что и другие инструменты сквозного тестирования, но с другим синтаксисом и внутренней реализацией. Если вы знакомы с Express (https://oreil.ly/6Asmr), то заметите сходство с написанием API-эндпоинтов. Это может быть удобно, если в вашей команде есть разработчики с сильной бэкенд-экспертизой, которые также работают с фронтендом и должны поддерживать покрытие кода тестами для своих изменений. Теперь можно заняться первым тест-кейсом — переходом на другую страницу:

```
it('переходит на страницу действий TestStore', (browser) => {
  browser
    .click('Actions')
    .assert.visible('Действия здесь')
    .assert.urlContains('/actions');
});
```

Синтаксис похож на тот, что используется в Selenium-тестах, поскольку Nightwatch также основан на спецификации W3C WebDriver (https://oreil.ly/HgVYS). Эта спецификация позволяет писать код, работающий во всех браузерах,

что критически важно для инструмента сквозного тестирования, так как тестирование должно проводиться в разных браузерах. WebDriver не требует компиляции с вашим кодом, поэтому, как и в других инструментах для такого тестирования, автоматизированные тесты выполняют действия так же, как если бы их совершал пользователь.

Следующий тест проверяет, корректно ли вызывается эндпоинт для получения данных о заказах:

```
it('загружает таблицу заказов', async (browser, client) => {
    client.assert.strictEqual(
        server.route.get('/v1/orders').calledOnce,
        true,
        'вызван один раз'
    );

    expect(browser.element.findByText('Dog stuff')).to.exist;
});
```

Здесь происходит обращение к вашему мок-API: проверяется, был ли вызван клиентом заданный маршрут, а также был ли загружен товар из заказа (смотрим на экране).

Следующий тест проверяет запрос на отправку формы:

```
it('отправляет успешный запрос на заказ', (browser, client) => {
  browser.element.findByLabelText('Имя').setValue('Ernest');
  browser.element.findByLabelText('Фамилия').setValue('Abcde');
  browser.element.findByLabelText('Количество').setValue('3');
  browser.element.findByLabelText('Email').setValue('e.abcde@ern.com');
  browser.element.findByLabelText('Пароль').setValue('B1gt3sTAcc0un!');
  browser.element.findByLabelText('Время контакта').setValue('2024-07-09T12:00');

  browser.element.findByText('Отправить').click();

  client.assert.strictEqual(
    server.route.post('/v1/orders').calledOnce,
    true,
    'вызван один раз'
  );

  expect(browser.element.findByText('Заказ успешно отправлен')).to.exist;

});
```

Формат здесь схож с другими инструментами: вы находите поле ввода и задаете ему значение. Затем находите кнопку «Отправить», кликаете ее и проверяете успешность выполнения POST-запроса.

Последний тест проверяет обработку невалидного email, чтобы убедиться в появлении сообщения об ошибке:

```
it('не отправляет неполный запрос на заказ', (browser) => {
  browser.element.findByLabelText('Имя').setValue('Ernest');
  browser.element.findByLabelText('Фамилия').setValue('Abcde');
  browser.element.findByLabelText('Количество').setValue('3');
  browser.element.findByLabelText('Email').setValue('e.abcde');
  browser.element.findByLabelText('Пароль').setValue('B1gt3sTAcc0un!');
  browser.element.findByLabelText('Время контакта').setValue('2024-07-09T12:00');
  browser.element.findByText('Отправить').click();

  const errorMessage = browser.element.findByText(
    "Пожалуйста, включите символ '@' в адрес электронной почты."
  );

  expect(errorMessage).to.exist;
});
```

Теперь вы можете запустить эти тесты, чтобы увидеть, как выглядит их выполнение: успешное или неуспешное. Особенность запуска тестов в Nightwatch заключается в том, что по умолчанию необходимо указывать путь к тестам в команде. Вы можете добавить скрипт в `package.json`, чтобы автоматизировать этот процесс для себя и коллег — как локально, так и в CI/CD-пайплайне. Вот команда для получения результатов тестов:

```
npx nightwatch nightwatch-tests
```

После выполнения вы увидите сообщения в терминале, а HTML-отчет о тестах сохранится в репозитории. Теперь, когда вы протестировали все эти инструменты и написали одни и те же четыре теста в каждом из них, можно сравнить их между собой.

Стоит отметить, что все эти пакеты для сквозного тестирования установлены как зависимости среды разработки (dev dependencies). Важно, чтобы инструменты тестирования не попали в продакшен-код. Не забывайте, что лишние пакеты в бандле замедляют работу приложения как на сервере, так и в браузере, а также повышают риск вредоносных атак. Любой из этих инструментов можно использовать для тестирования API, если вам нужно применять одинаковые средства тестирования и на фронтенде, и на бэкенде.

Сравнение пакетов

Итак, вы увидели, как запускать сквозные тесты с помощью разных инструментов, как настраивать среды, как инструменты работают в CI/CD-пайплайне и помогают проверять функциональность с точки зрения пользователя. На этом этапе можно плотно поработать с продуктовой командой, чтобы уточнить требования и подходы к обработке пограничных случаев. Есть правило: если вам сложно придумать содержательные названия для тестов, скорее всего, вы пишете не очень хорошие тест-кейсы. Учитывайте это, когда делаете ревью пул-реквестов от других разработчиков и добавляете новые тест-кейсы.

При сравнении Cypress, Playwright и Nightwatch ключевой метрикой является время, затрачиваемое разработчиками на написание и поддержку тестов. Иногда тот или иной инструмент команде освоить проще, чем другие, и это веская причина выбрать именно его. Если команда пишет тесты в каком-то конкретном инструменте на 10 % быстрее, то в долгосрочной перспективе это сэкономит массу времени, ведь по мере изменения кода и появления новых функций придется обновлять и тесты.

Время выполнения тестов в вашем CI/CD-пайплайне — еще одна важная метрика, на которую стоит обратить внимание. Размер пакета может влиять на этот показатель, поскольку перед запуском тестов необходимо установить сам пакет. Однако если время выполнения тестов для вашего приложения примерно одинаково для всех пакетов, стоит рассмотреть другие метрики.

Как и при выборе любых пакетов, вам необходимо определить, какой из них предоставляет наиболее удобную документацию для вас и вашей команды. Поддержка сообщества также играет ключевую роль, поскольку от нее зависит скорость решения проблем и доступность помощи при работе с особенностями конкретного пакета. Рекомендуется изучить раздел GitHub Issues для этих пакетов, чтобы оценить, насколько эффективно разрешаются запросы.

Пирамида тестирования

Пирамида тестирования (testing pyramid) — это полезная концепция, помогающая определять, как и когда внедрять различные типы тестов, например модульные или сквозные. Вот краткое пояснение от Итана Брауна.

> Общий принцип звучит так: сквозных тестов должно быть меньше всего, поскольку они очень дорогие. Инструменты вроде Cypress, на мой взгляд, постепенно меняют эту парадигму, но сам принцип остается неизменным: усилия, вкладываемые в технологию, должны быть пропорциональны ее стоимости (в деньгах или человеко-часах) и ценности. Можно много спорить о том, насколько Cypress меняет соотношение стоимости и ценности для сквозных инструментов, но сложно оспаривать верность самого принципа.

Cypress, вероятно, самый известный инструмент для сквозного тестирования в экосистеме JavaScript, поэтому его часто выбирают первым — многие разработчики уже с ним знакомы. Но если вы и команда предпочитаете Playwright или Nightwatch, то они тоже широко распространены. Учитывайте использование выбранного пакета в долгосрочной перспективе — от этого будет зависеть, как быстро адаптируются новые разработчики в проекте, поскольку выбранный инструмент в вашем процессе разработки станет стандартом.

Заключение

Теперь у вас есть все необходимое для тестирования приложения целиком — от фронтенда до бэкенда и, возможно, частично базы данных. Обычно этим занимаются QA-инженеры, но если в команде их нет, вы можете применить свои навыки для организации тестирования. У вас должно уже сформироваться свое мнение на тему того, как писать тесты, какие инструменты использовать и почему стоит тратить время на сквозные тесты в дополнение к модульным.

Учитывайте, что модульные тесты нужны для проверки кода в конкретных сценариях, а сквозные имитируют действия пользователя, чтобы убедиться, что части приложения работают как ожидается. Это целостный взгляд на процессы, где вас не должен волновать стоящий за этими процессами код.

Сквозные тесты также можно показывать продуктологам и стейкхолдерам, чтобы они видели, как тесты выполняются и какую ценность добавляют. Такая демонстрация может инициировать обсуждение будущих направлений работы, когда заинтересованные стороны увидят автоматизацию в действии. Учитывайте, что написание и поддержка таких тестов требуют много времени, поэтому, возможно, не стоит браться за них немедленно. Но и затягивать тоже не стоит, иначе накопится много непротестированных функций.

ГЛАВА 25

Организация развертываний

Теперь, когда у вас настроено интеграционное тестирование, пора решить, когда и как выполнять развертывания. Каждое развертывание для кого-то важно:

- для разработчика, занимающегося изменениями;
- для QA-инженера, проводящего тестирование функций и регрессионное тестирование;
- для стейкхолдеров, оценивающих результат;
- для конечных пользователей.

Каждое развертывание влияет и на что-то еще. Когда вы вносите изменения в бэкенд, это так или иначе скажется на фронтенде. Изменения во фронтенде затрагивают всех, кто взаимодействует с продуктом через эту часть приложения.

Наличие стратегии и понимание сроков развертывания критически важны, чтобы избежать неожиданностей. Внедрение изменений влияет на организацию и репутацию вашей команды в плане качества и надежности. Это лишь часть аспектов, о которых необходимо задуматься. Может показаться, что работа завершена, когда все изменения интегрированы в код, но на самом деле это лишь начало одной из наиболее заметных частей вашей работы.

В этой главе мы рассмотрим:

- обновления только фронтенда или только бэкенда;
- сине-зеленые развертывания (blue-green deploy);
- канареечные развертывания (canary deploy);
- стратегии отката изменений.

Подобно тому как вам приходилось учитывать множество «закулисных» соображений при первоначальной разработке всего стека, то же самое придется делать и при развертывании — на этапе планирования нужен целостный взгляд. При создании приложения вы проверяли, чтобы все было совместимо с вашими инструментами и технологиями. И сейчас необходимо думать об общем влиянии на продукт, задействованных командах и пользователях. Важно найти баланс между скоростью и рисками.

Развертывание обновлений только для фронтенда или бэкенда

Часто бывает так, что бэкенд и фронтенд выпускают обновления по разным графикам. Одни изменения потребуют развертывания только одной части приложения, другие — координации по всему стеку. Важно учитывать взаимодействие частей стека друг с другом и с другими сервисами. Сроки развертывания особенно важны, когда известно, что нужно дождаться обновления с одной или другой стороны стека.

Работая над функционалом бэкенда, необходимо понимать, как ваши изменения затрагивают другие части системы. Одна из задач — проверить, не изменят ли обновления данные, которые ожидает фронтенд. Если изменят, значит, вы вносите критическое изменение, которое следует учитывать дополнительно. Также необходимо проверить, изменяете ли вы данные, отправляемые стороннему сервису. Это может быть связано с обновлением самого сервиса или рефакторингом кода для повышения его управляемости. Кроме того, важно оценить, повлияют ли изменения на взаимодействие приложения с компонентами инфраструктуры, такими как база данных.

Убедитесь, что ваши задачи (jobs) не пострадают от этих изменений. Иногда рефакторинг может привести к неожиданным побочным эффектам. Например, обновление типов или реструктуризация папок может сломать функцию, которую вы не тестируете. Модульные и интеграционные тесты помогают выявить многие регрессионные ошибки, но некоторые вещи все равно могут ускользнуть. Бэкенд обычно затрагивает больше частей системы, чем фронтенд, и здесь особенно полезными становятся документация и архитектурные диаграммы. Благодаря им при внесении изменений вы точно знаете, как все взаимосвязано и что может быть затронуто.

Фронтенд, как правило, можно развернуть, не влияя на другие части системы. Однако при его развертывании важно убедиться, что бэкенд и другие сервисы готовы. Если развернуть изменения во фронтенде до обновления бэкенда, есть риск сломать часть функциональности для пользователей. Также стоит учитывать возможные пересечения с изменениями других разработчиков, особенно если несколько человек работают с одними и теми же файлами. Это может привести к перезаписи функционала или дизайнов.

Бывает, что дизайн меняется для одной части приложения, и это создает визуальную несогласованность с остальным интерфейсом. Поэтому команде важно учитывать последствия изменений общих компонентов. Например, обновление компонента под требования одного раздела может нарушить верстку в другом. Перед развертыванием проверьте все ссылки на общие компоненты, чтобы убедиться в отсутствии нежелательных побочных эффектов.

При раздельном развертывании всегда сначала убедитесь, что фронтенд и бэкенд корректно работают в среде разработки. Полезно версионировать релизы

фронтенда и бэкенда, используя немного разные номера. Например, бэкенд может быть на версии 2.0.0, а фронтенд — на 1.0.0. Эти номера указываются в файле `package.json` (https://oreil.ly/HMP5q) для каждого проекта. Также рекомендуется обозначать версию бэкенда, от которой зависит фронтенд.

Вы и команда разработчиков — первая линия QA, поэтому крайне важно уделять внимание деталям. Это особенно актуально, когда вы долго фокусировались на одном аспекте приложения. Используйте чек-листы или автоматизацию, чтобы помочь команде заметить то, что могло ускользнуть из-за погружения в детали одной части стека.

Стратегии развертывания

Работая над проектами в разных командах и организациях, вы столкнетесь со множеством подходов к развертыванию. В крупной компании процессы будут кардинально отличаться от процессов в компании среднего размера или стартапов. Здесь важно учитывать:

- когда масштабировать инфраструктурные ресурсы;
- как планировать работу над новыми функциями;
- какие приоритеты у команды разработчиков.

На этом этапе вы меньше сосредоточены на коде и технической реализации, а больше взаимодействуете с продуктовой командой.

Даты релизов

Ключевой момент для согласования с продуктовой командой — конкретные даты релизов. Это дедлайны, к которым команда разработчиков должна подготовить функционал и исправления для продакшена. Обычно продуктовая команда согласует дату релиза с заинтересованными сторонами (стейкхолдерами), а затем обсуждает ее с вашей командой разработчиков. Здесь важно убедиться, что команда полностью понимает все требования релиза.

Запросите дизайны и спецификации и выделите время на их обсуждение в команде, чтобы поднять все технические вопросы. Если со стороны продуктовой команды чего-то не хватает, дайте обратную связь и сообщите, что не сможете подтвердить дату релиза, пока не получите все необходимые детали для работы. Покажите продуктовой команде, чего именно не хватает, и совместно заполните пробелы. Выполнять задачу с поверхностым и нечетким ТЗ — значит, создавать для команды дополнительный стресс: в последний момент придется в спешке дописывать код, чтобы уложиться в срок.

Продуктовая команда будет активно взаимодействовать с командой разработчиков, чтобы понять технические требования и донести их до стейкхолдеров. Обсуждая

требования с командой, установите внутренний дедлайн, известный только разработчикам. Важно помнить: дата релиза — это срок, к которому продуктовая команда должна получить функционал в продакшене. Это означает, что весь код уже объединен, протестирован и работает без ошибок. Полезно взять дату релиза и двигаться от нее назад, чтобы оценить, успеете ли вы с командой внести все изменения и протестировать их, как показано на рис. 25.1.

Поддерживайте постоянную коммуникацию между командой разработчиков и продуктовой командой

Ваша роль теперь, помимо прочего, состоит в том, чтобы взаимодействовать с техническими и нетехническими командами. Такое взаимодействие включает в себя поддержание репутации команды разработчиков. Хотя этим обычно занимается менеджер или техлид, иногда ответственность ложится на вас, особенно если ваш менеджер не обладает техническими знаниями. Важно, чтобы вас не воспринимали как команду, которая постоянно срывает сроки или выпускает в продакшен некорректные изменения или неполные требования, даже если это не ее вина.

При разработке стратегии развертывания предусмотрите временной буфер для команды на случай непредвиденных обстоятельств. Подходите к утверждению пул-реквестов на развертывание с оптимистичной осторожностью и тщательно тестируйте их перед передачей в QA. Это дополнительный шаг, но он окупается, когда вы находите ошибки раньше других.

Цель — добиться непрерывного цикла релизов, где каждое небольшое изменение быстро проходит QA и попадает в продакшен. Так вам не придется делать крупные релизы одновременно. График релизов в продакшен зависит от размера организации: чем больше команд задействовано в разных частях процесса, тем сложнее синхронизация. В крупных компаниях в релизах могут участвовать юридические отделы, чтобы проверить условия использования новых функций. Также может потребоваться более активная работа с командой DevOps, чтобы соблюсти дату релиза.

В итоге у вас может получиться всего 3–4 релиза в продакшен за год, несмотря на активную работу над функциями и частые выкатывания в стейджинг-среду. В небольших компаниях релизы в продакшен могут происходить каждые несколько дней или недель. В организациях среднего размера и стартапах вы, вероятно, будете делать больше самостоятельно и взаимодействовать с гораздо меньшим количеством команд, что ускорит процесс выпуска функций. Однако даже в таких организациях со временем может сложиться ситуация, когда релизы в продакшен будут происходить раз в несколько месяцев — просто потому, что по мере роста компании не хватает людей, чтобы справляться со всем объемом работы.

Март 2024 г.

Воскресенье	Понедельник	Вторник	Среда	Четверг	Пятница	Суббота
25	26 Спецификации и дизайны готовы: 26.02.2024	27 Начало разработки: 27.02.2024	28	29	01	02
03	04	05 Первые функции проверены и готовы для QA: 05.03.2024	06 Первые функции развернуты в стейджинг-среде QA: для 06.03.2024	07	08	09
10	11 QA дает обратную связь: 11.03.2024	12	13	14 Доработки по замечаниям QA развернуты в стейджинг-среде: 14.03.2024	15	16
17	18	19 Демо для продуктовой команды, QA и дизайнеров для дополнительной обратной связи: 19.03.2024	20	21	22 Доработки по всем замечаниям: 22.03.2024	23
24	25	26	27 Дополнительные демо: 27.03.2024	28	29 Подготовка к релизу в продакшен: 29.03.2024	30
31	01	02 Финальные проверки перед релизом в продакшен: 02.04.2024	03	04 Дата релиза: 04.04.2024	05	06

Рис. 25.1. Пример графика релизов

При анализе требований к функционалу обновляйте техническую документацию, фиксируя все условия, которые должно учитывать приложение. Это пригодится при обсуждении будущих функций. Также узнавайте дорожную карту продукта, чтобы совместно с командой расставлять приоритеты задач. Продуктовая команда должна контролировать приоритизацию, но вам важно согласовывать с ней сроки, особенно если технические нюансы влияют на график.

В некоторых отраслях (например, в рекламе и финансах) существуют периоды года, когда релизы в продакшен невозможны, потому что клиенты тратят больше всего денег. Такие периоды называются заморозками (freeze period), и обычно они приходятся на крупные праздники. Если вам нужно поработать над техническим

долгом и реализовать новые функции, попробуйте перенести работу с техдолгом на эти периоды — так вы сможете сосредоточиться на релизах.

Версионирование релизов

Еще одна стратегия, которая поможет в отладке, — версионирование релизов ваших приложений, как упоминалось ранее в главе. Это особенно актуально для бэкенда, где изменения в API могут вызвать больше проблем, чем на фронтенде. При версионном релизе бэкенда текущая версия обычно поддерживается какое-то время. Это дает фронтенду и другим зависимым приложениям возможность перейти на новую версию без немедленного нарушения текущей функциональности. Такой подход является формой плавной деградации (graceful degradation).

Когда наступает дата деградации версии, у вас есть два варианта: оставить старую версию доступной с пометкой о прекращении поддержки и необходимости обновления для пользователей либо полностью удалить старую версию. Это решение необходимо согласовать с продуктовой командой и стейкхолдерами, особенно если оно потребует дополнительных действий от команды разработчиков вне обычного цикла создания функций.

Следование семантическому версионированию (semantic versioning, https://semver.org) помогает всем участникам понимать тип изменений в новой версии. Семантическое версионирование с помощью числовых обозначений указывает на критические изменения, обратно совместимые изменения или исправления ошибок. Честно говоря, во многих организациях эти правила соблюдаются весьма свободно — некоторые команды выпускают версии типа v0.193.36, так и не дойдя до v1.0.0. В идеале старайтесь максимально придерживаться семантического версионирования. CHANGELOG становится критически важным при версионных релизах — он позволяет быстро увидеть различия между версиями и помогает в отладке других приложений.

Существует множество инструментов для автоматизации семантического версионирования через git-хуки или процесс релиза. Среди них: release-please (https://oreil.ly/U_3Xi), commit-and-tag-version (https://oreil.ly/O5zze) и release-it (https://oreil.ly/H1GhV). Следование принципам Conventional Commits (https://oreil.ly/kIuoy), как будет показано в git-хуках в главе 27, упрощает переход команды на семантическое версионирование. Также можно использовать команду `npm-version` (https://oreil.ly/rQ3te) для обновления версии пакета. Все эти инструменты обновляют версию в `package.json` и `package-lock.json`. Дополнительно можно применять Git-теги (https://oreil.ly/Hexzf) для сохранения артефактов каждой версии приложения.

На фронтенде версионирование интерфейса может служить инструментом для отладки и решения проблем в продакшене. Иногда развертывания завершаются без явных ошибок, но изменения не попадают на сервер, как ожидалось. Без указания версии в интерфейсе может потребоваться значительное время, чтобы определить, что корневая причина — неудачное развертывание. Это также полезный

инструмент для команды поддержки, когда пользователи обращаются с проблемами. Фронтенд обычно не поддерживает плавную деградацию, поскольку для всех пользователей должна быть доступна только одна версия интерфейса.

Версионирование фронтенда обычно означает отображение версии приложения в компоненте, который всегда рендерится на странице. В примере приложения, которое вы создали, это может быть навигационная панель. В других приложениях номер версии может находиться в футере или другом малозаметном месте. Обычно пользователям не нужно обращать на это внимание, но номер может потребоваться, если им будет нужно обратиться в поддержку. Номер версии должен отображаться в интерфейсе независимо от того, авторизован пользователь или нет.

Вы можете наблюдать подобное версионирование в сторонних сервисах. Многие из них используют семантическое версионирование и плавную деградацию для выпуска изменений. Они объявляют сроки обновлений, чтобы пользователи успели адаптировать свои системы к новой версии. Например, если текущая версия вашего сервиса — v1.3, а новая — v2.0, вы понимаете, что речь идет о критическом изменении. В таком случае вам стоит как можно скорее обновить версию сервиса в своих приложениях и провести тестирование.

Сине-зеленые развертывания

В ситуациях, когда ваше приложение обрабатывает большой объем трафика и вы не можете позволить себе простои для всех пользователей, отлично подходит так называемое сине-зеленое развертывание. Термины «синий» (blue) и «зеленый» (green) обозначают окружения, в которых работает приложение. Например, в синем окружении может работать старая версия приложения, а в зеленом — новая, хотя цвета можно менять местами. Это позволяет постепенно перенаправлять трафик со старой версии приложения на новую. Таким образом, в продакшене одновременно работают две версии одного приложения, получая разный объем трафика.

Если одним из ключевых показателей вашего приложения является время бесперебойной работы, вам следует обсудить этот подход со всеми заинтересованными сторонами, так как он потребует дополнительных усилий. Вы можете начать с перенаправления 10 % трафика в зеленое окружение и проверить, насколько хорошо работают изменения. По мере подтверждения корректной работы всех компонентов можно постепенно увеличивать долю трафика в зеленом окружении. После каждого увеличения трафика анализируйте данные мониторинга: отслеживайте динамику ошибок и адаптацию использования ресурсов.

В конечном счете вы достигнете точки, когда весь трафик будет перенаправлен в зеленое окружение. Синее окружение можно оставить в режиме ожидания на случай возникновения проблем — это позволит быстро вернуть трафик обратно. Как только вы убедитесь, что зеленое окружение стабильно обеспечивает требуемое время бесперебойной работы, синее окружение можно отключить в целях оптимизации затрат. Команда DevOps может использовать эту схему как шаблон

для будущих обновлений, меняя роли синего и зеленого окружений после каждого развертывания, — это упрощает процесс для команды.

Сине-зеленые развертывания полезны не только для обеспечения бесперебойной работы. Они позволяют легко проводить A/B-тестирование и наблюдать за взаимодействием части пользователей с новыми функциями перед массовым релизом. Это дает продуктовой команде исследовательские данные для формирования будущей дорожной карты. Вы можете отслеживать ключевые метрики, чтобы выявить значимые различия между двумя версиями приложения. Кроме того, такой подход позволяет тестировать изменения непосредственно в продакшене. Все команды могут выполнить финальное тестирование для проверки работоспособности перед переключением трафика, поскольку изменения уже развернуты в рабочей среде.

Однако у этого подхода есть и недостатки. Настройка окружений и ресурсов для сине-зеленых развертываний требует значительных временных и трудозатрат. Это дорогая стратегия, так как приходится поддерживать две продакшен-среды с идентичными ресурсами. Требуется масштабное тестирование для гарантии одинаковой работы обеих сред. Также могут возникнуть проблемы с синхронизацией данных между средами, особенно если в одной из них есть изменения схемы данных, отсутствующие в другой. Контейнеризация может помочь решить эти проблемы; подробнее об этом рассказывается в главе 26.

Это справедливо для любых внешних сервисов, поскольку вы будете использовать одни и те же учетные данные для подключения из разных сред. Вам необходимо проверить лицензионные соглашения с вашими сервисами, чтобы убедиться, что это возможно. Также проверьте наличие косвенных взаимодействий между средами через внешние сервисы. Это может привести к неожиданным изменениям данных или конфигурации, что сделает обе среды ненадежными. Иногда существует риск утечки данных между двумя средами. Например, у вас могут быть кэшированные DNS-записи, которые перенаправляют синий фронтенд на зеленый бэкенд, и ситуация может усугубиться, если DNS-записи обновляются в разное время из-за географического расположения. Сине-зеленые развертывания работают как для бэкенд-, так и для фронтенд-приложений, но важно, кроме преимуществ, учитывать и риски.

Канареечные развертывания

Канареечные развертывания похожи на сине-зеленые, но с небольшими отличиями. В канареечных развертываниях вам не нужно сохранять две полноценные продакшен-среды. Этот подход использует фича-флаги (feature flag) для постепенного выпуска изменений. Таким образом, вы можете развернуть изменения в продакшен-среде, но они будут скрыты от пользователей до тех пор, пока вы не предоставите им соответствующий доступ. Включение и отключение частей кода потребует дополнительной настройки в приложении, поэтому вам придется совместно с командой продумать оптимальный способ это осуществить.

Канареечные развертывания позволяют запускать новый функционал параллельно с существующим, а не изолировать их в отдельном окружении. Можно представить этот подход как постепенное раскрытие функции в продакшене, вместо того чтобы держать новую версию в одном окружении, а старую — в другом. Этот подход похож на прогрессивную доставку (progressive delivery) (https://oreil.ly/QqVSi) или поэтапное «выкатывание» (staged rollout) (https://oreil.ly/X6kZx).

Команде разработчиков придется принимать дополнительные архитектурные решения для реализации фича-флагов. Необходим устойчивый механизм управления этими флагами для разных групп пользователей.

При этом подходе легко забыть полностью включить функцию, из-за чего команда поддержки начнет получать отчеты о проблемах. Или можно слишком рано отключить фича-флаг, открыв пользователям незавершенные функции. Вы можете создать собственный инструмент для управления фича-флагами или использовать готовые решения: FeatureFlags (https://featureflags.io), LaunchDarkly (https://oreil.ly/-f37f), Unleash (https://getunleash.io) и PostHog (https://oreil.ly/n9BV8). В любом случае обязательно документируйте все флаги в приложении, их текущий статус выкатывания и группы пользователей с доступом к ним.

Стоит завести тикет с напоминанием о необходимости периодической очистки флагов функций, чтобы вы и команда не потеряли контроль над актуальными настройками. Использование флагов дает несколько преимуществ: повышает прозрачность предстоящих изменений для всей команды, приучает разработчиков создавать стабильные пул-реквесты и стимулирует более вдумчивый подход к проектированию надежной системы управления флагами.

Не стоит путать канареечные развертывания (canary deployments) с канареечными релизами (canary releases). Канареечный релиз — это способ тестирования ранней версии приложения с пользователями, которые любят пробовать новые инструменты первыми. Часто такой подход встречается в проектах с открытым кодом, когда выходят нестабильные версии с отдельной нумерацией, отличающейся от стабильных релизов. Крупные компании, например Google с канареечными сборками Chrome, также используют эту практику.

Это лишь несколько стратегий, которые можно применять для выпуска изменений на фронтенд и бэкенд для пользователей. Однако даже после успешного развертывания нужно быть готовым к возможным проблемам и иметь планы по их устранению.

Стратегии отката изменений

Несмотря на тщательное планирование и тестирование, в продакшене могут возникать проблемы. Если до последнего (перед сбоем) развертывания все работало стабильно, логично предположить, что ошибка связана с изменениями в коде. Хотя это не всегда так, но тем не менее код — это область, которую вы контролируете

в наибольшей степени. Поэтому важно заранее подготовить планы действий на случай непредвиденных ситуаций.

Развертывание предыдущих версий

Одна из стратегий отката — повторное развертывание предыдущей версии артефакта приложения. Такое развертывание упрощается, если вы используете Git-теги для версий релизов или храните предыдущие сборки в облачном хранилище. Стратегии сине-зеленого или канареечного развертывания также облегчают откат: вам не потребуется вносить изменения в код или создавать новые пул-реквесты для запуска пайплайна.

В этом случае важно взаимодействовать с командой DevOps. Они могут выполнить откат вручную или предоставить интерфейс для выбора нужной версии в продакшене. Это быстрый способ снять нагрузку с разработчиков, пока они ищут корневую причину проблемы.

Отмена или сброс пул-реквестов

Еще одна стратегия — отменить пул-реквест для релиза. В нашем примере это означает удаление кода, который был объединен в основную ветку и вызвал развертывание в пайплайне. Вам потребуется создать новый пул-реквест, нацеленный на основную ветку, что запустит новое развертывание для возврата к предыдущей рабочей версии кода. Это можно сделать несколькими способами с помощью команд Git, которые здесь лишь кратко упоминаются, поскольку подробному обсуждению этой темы посвящена глава 28.

Один из способов — использовать команду `git revert` (https://oreil.ly/So9W5) с указанием конкретного ID коммита, как показано на рис. 25.2. Хотя эта команда может показаться заманчивой, не используйте ее, если не понимаете, о чем говорится в документации по откатам (https://oreil.ly/5kxdh). Для глубокого изучения Git рекомендуется книга *Version Control with Git* (Prem Kumar Ponuthorai, Jon Loeliger (O'Reilly))[1]. Проще говоря, `git revert` удаляет изменения только в рабочей области, не затрагивая историю Git, — это похоже на локальную отмену. Однако если вы не эксперт в Git, лучше избегать этой команды.

Если вам нужно отменить несколько коммитов, следует рассмотреть использование команды `git reset` (https://oreil.ly/HvFAn). Эта команда отменит все изменения, произошедшие после указанного коммита. Таким образом, `reset` отменяет коммиты кумулятивно, а не только указанный коммит. Выбранный вами вариант `reset` крайне важен. Если вы выполните `git reset --soft`, это отменит коммит, но оставит изменения в состояниях стейджинг-среды, как показано на рис. 25.3. Это позволит вам повторно просмотреть изменения и решить, как с ними поступить.

---

[1] На русском языке: Понуторай Прем Кумар, Лолигер Джон. Git: контроль версий. — Sprint Book.

```
milecia@Milecias-MacBook-Pro-2 dashboard-ui % git log -3
commit 0cb0ca536ad7304e0709324f8a160e05ac1d0c79 (HEAD -> ch-15, origin/ch-15)
Author: flippedcoder <milecia.mcgregor@aol.com>
Date:   Mon Jan 15 09:48:16 2024 -0600

    added backend requests and axios config

commit 1e6e69456dac007cd32ebe71b75e62b6011fb286
Author: flippedcoder <milecia.mcgregor@aol.com>
Date:   Sun Jan 7 20:25:31 2024 -0600

    created .env file

commit feb094644379986dfc3cbd540fa4f3e49eb9acf7
Author: flippedcoder <milecia.mcgregor@aol.com>
Date:   Sun Jan 7 20:17:48 2024 -0600

    wrapped App in QueryClientProvider
milecia@Milecias-MacBook-Pro-2 dashboard-ui % git revert 0cb0ca536ad7304e0709324f8a160e05ac1d0c79
Removing src/axios.config.ts
[ch-15 f32745b] Revert "added backend requests and axios config"
 2 files changed, 29 insertions(+), 43 deletions(-)
 delete mode 100644 src/axios.config.ts
```

Рис. 25.2. Пример использования `git revert`

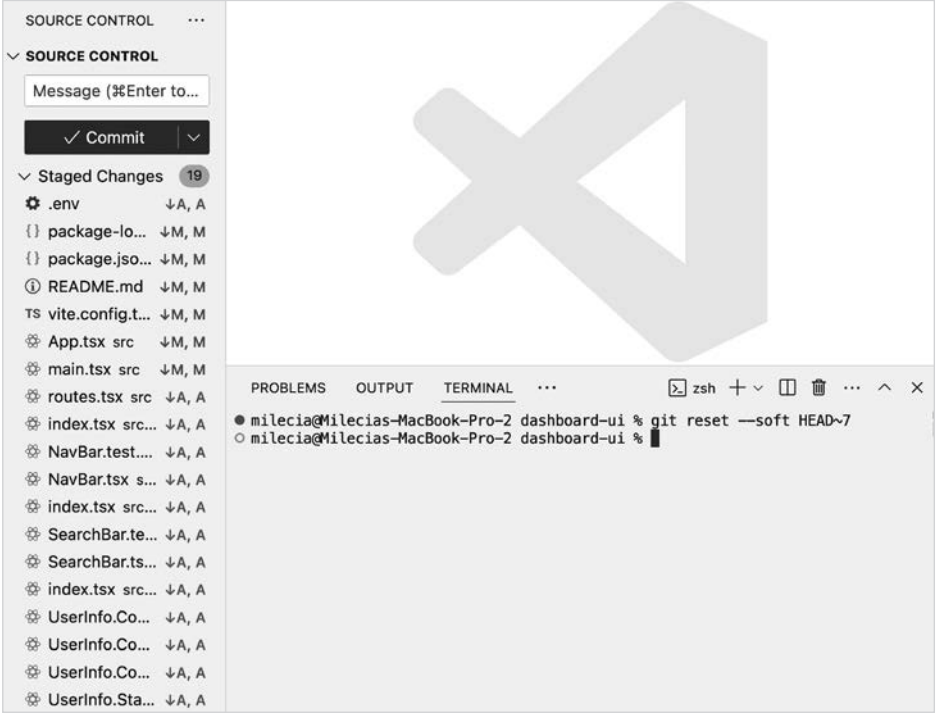

Рис. 25.3. Пример использования `git reset --soft`

Если выполнить `git reset --mixed`, изменения будут отменены и выведены из состояний стейджинг-среды, как показано на рис. 25.4. В результате отмененные

коммиты будут выглядеть как локальные изменения. Другой вариант — выполнить `git reset --hard`. Это полностью удалит изменения, включая их состояние в стейджинг-среде, как показано на рис. 25.5. Будьте осторожны с `hard reset`, так как изменения будут безвозвратно утеряны.

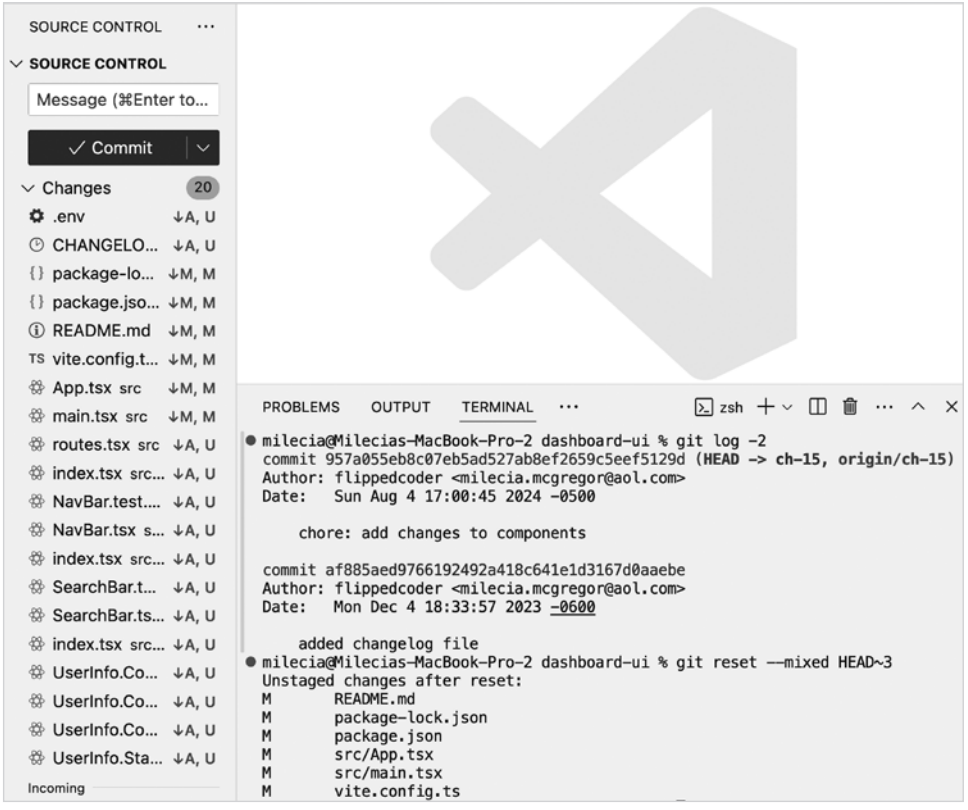

Рис. 25.4. Пример использования `git reset –mixed`

Многие разработчики путают команды `revert` и `reset`, поскольку часто требуется отменить только последний коммит. В этом сценарии обе команды работают хорошо. Однако, как только нужно отменить более одного коммита, становится критически важным понимать разницу между командами и используемыми опциями.

Самые простые и легкие подходы — это повторное развертывание старого артефакта, переключение трафика на синее или зеленое окружение либо повторный релиз предыдущей версии с тегом. Необходимо уверенно владеть командами Git и понимать причины их использования, иначе можно создать еще больше проблем. Уделите время экспериментированию с этими командами, чтобы увидеть, как они изменяют код.

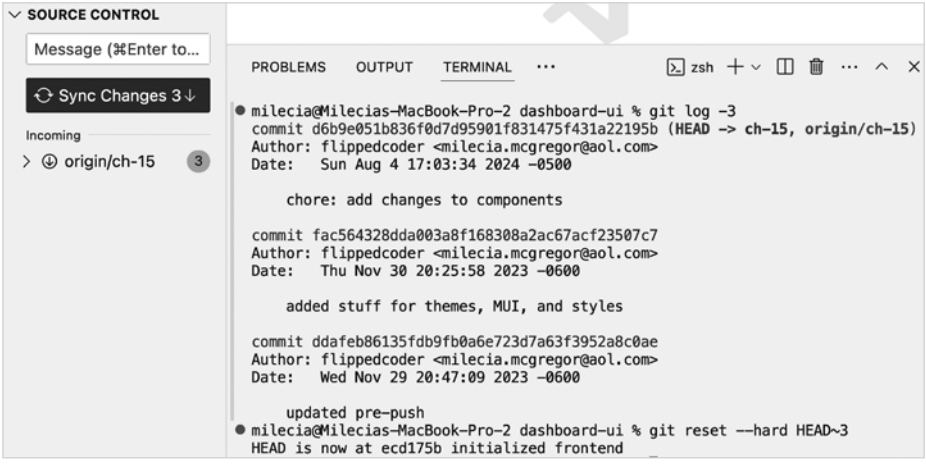

Рис. 25.5. Пример использования `git reset –hard`

Развертывание хотфикса

Это скорее стратегия продвижения вперед, поскольку вы ничего не откатываете. Команда разработчиков максимально быстро ищет первопричину, исправляет код и создает новый пул-реквест в основную ветку. Такой подход бывает оправдан в ситуациях, когда инфраструктура не настроена для откатов, а репозиторий не поддерживает команды `revert`. В этом случае единственный вариант — двигаться вперед. Это может создавать дополнительное давление на рабочую группу, особенно если простой или ошибки приносят убытки пользователям и компании.

Реалистичный подход к хотфиксам

Итан Браун добавляет важное замечание о хотфиксах.

> Вполне разумно и реалистично создать «протокол хотфиксов», который может включать сниженные требования к ревью, тестированию и проверкам, — при условии, что последним шагом процесса будет «выполнить все стандартные процедуры ревью, тестирования и проверки сразу после устранения кризиса». На этом же этапе приоритетом является выпуск хотфикса, который превалирует над любой другой работой.
>
> В зависимости от реакции на давление можно выделить два типа людей: тех, кто честно признает ошибки, и лжецов. Давайте не будем стыдить первых и позволять вторым избегать ответственности. Создайте процесс, соответствующий ситуации, и предусмотрите в нем механизм восстановления пропущенных этапов после того, как пожар будет потушен.

Если вам приходится работать по стратегии хотфикса, начните с изучения логов и последнего коммита — скорее всего, проблема кроется именно там. В такой ситуации особенно пригодится ваш опыт. Сохраняйте спокойствие и следуйте стандартному процессу отладки, как в любом другом окружении. Хотфикс не проходит тот же процесс QA, что обычные исправления багов или развертывание функций. На этом этапе именно вам и команде разработчиков нужно протестировать конкретную проблему. Организуйте встречу, чтобы несколько человек могли совместно проверить исправление. Подключив к встрече QA и продуктовую команды, вы сможете проверить и другие функции, чтобы быть полностью уверенными, что хотфикс не создаст новых проблем.

Заключение

В этой главе мы рассмотрели несколько подходов, которые можно применять при развертывании. Это далеко не все возможные подходы, поэтому смело проявляйте творчество и адаптируйте их под свои нужды. Возможно, лучше всего вам подойдет вариант, где вы их скомбинируете. Обязательно тесно сотрудничайте с командами DevOps и QA — успех ваших релизов напрямую зависит от инфраструктуры, на которой выполняется код, и тщательности тестирования. Держите их в курсе, чтобы все участники процесса понимали план действий и знали график релизов.

Развертывание не должно превращаться в стресс для команды, если вы заранее предусмотрели время на исправление багов и подготовку облачных ресурсов. Постоянная и частая коммуникация с продуктовой командой и стейкхолдерами поможет избежать неожиданностей для всех. Сообщайте о возникающих проблемах, даже если они на первый взгляд не повлияют на сроки. Держать всех в курсе — важная часть любой вашей стратегии. Это окупится в виде большего доверия и свободы действий для вас и вашей команды.

ГЛАВА 26

Проблемы интеграции

Вы успешно развернули фулстек-приложение в продакшене, и все, кажется, работает как надо. Развертывание приложения — это серьезное достижение, и теперь у вас есть прочная основа для дальнейшей работы. На этом этапе можно приступить к поиску областей для улучшения. Именно сейчас можно провести стресс-тестирование приложения, чтобы проверить, насколько хорошо связаны фронтенд и бэкенд, а также как ведут себя сторонние сервисы в реальных условиях.

Когда вы и другие команды начнете тестировать приложение в продакшене, неизбежно возникнут определенные проблемы. Некоторые из них окажутся неожиданными, несмотря на тщательное планирование. Например, кто-то может обнаружить, что у него нет доступа к определенной странице или что на фронтенде появился неожиданный «глюк». Хотя вы и команда сделали все возможное, чтобы протестировать все в стейджинг-среде, некоторые вещи просто ведут себя по-другому. Это момент, когда нужно устранять те проблемы, которые невозможно было выявить в других средах.

В этой главе мы разберем:

- проблемы фронтенда и бэкенда;
- аспекты, которые стоит проверить в сторонних сервисах;
- управление данными и безопасностью в продакшене;
- контейнеризацию вашего приложения.

Этот этап жизненного цикла разработки требует упорства и настойчивости, ведь вы уже так близки к завершению! Эти последние проблемы интеграции помогут вам, команде и всем остальным убедиться, что вы можете автоматизировать развертывания с уверенностью в их стабильной работе. Раньше вы разрабатывали и тестировали каждый слой изолированно, поскольку должны были сосредоточиться на разработке. Однако некоторые проблемы, возникающие на текущем этапе, будет сложнее выявить, чем любые предыдущие, поскольку их причиной может быть что угодно в вашем окружении.

Проблемы фронтенда и бэкенда

Вы протестировали фронтенд и бэкенд по отдельности, у вас есть модульные тесты для них, а также несколько сквозных тестов. Теперь нужно начать проверять вещи,

которые помогут по максимуму сохранить удобство пользования приложением. Одна из таких вещей — двойная проверка обработки ошибок. Рассмотрим случай, когда вы протестировали все в стейджинге, а теперь в продакшене отсутствует переменная окружения. Как отреагирует приложение? Или что произойдет, если развертывания фронтенда и бэкенда рассинхронизируются из-за нового члена команды, который еще не до конца разобрался в процессе?

Понимание собственного графика развертывания определяет, когда и как выпускаются новые функции и происходят исправления ошибок. Зачастую правильно разворачивать первым именно бэкенд, за исключением случаев, когда в API вносятся критические изменения. В таких случаях необходимо согласовать действия фронтенда и бэкенда, чтобы минимизировать простой для пользователей, или выполнить развертывание в непиковые часы.

Поскольку команда работает над функционалом асинхронно, важно учитывать изменения в коде, внесенные другими разработчиками. Иногда изменения незаметно мержатся в ветки, что может создать проблемы для всех. В главе 27 я расскажу о стратегиях ветвления в Git, но всегда проверяйте, какой код находится в ветке, прежде чем начинать процесс развертывания. Несколько функций могут мержиться в одну ветку и иметь несовместимые элементы, для устранения которых требуется время. Обсуждайте планы развертывания заранее, чтобы все были в курсе происходящего за пределами их собственной работы.

Выполняли ли вы тестирование в разных браузерах и на разных устройствах? Иногда старые версии браузеров могут нарушить верстку фронтенда, поскольку не поддерживают используемые вами функции. Этот момент легко упустить из виду на этапе препродакшен-тестирования, так как всем хочется побыстрее сделать релиз изменений. Также необходимо убедиться, что реализован адаптивный или отзывчивый дизайн, учитывающий просмотр приложения на экранах устройств разного размера. Особое внимание стоит уделить пограничным случаям взаимодействия пользователя.

Еще один важный нюанс — обработка временных зон в продакшене, особенно для запланированных задач. Разница в миллисекундах из-за расположения серверов может привести к проблемам, например, задача выполнится сегодня, а не завтра, или наоборот. Это также может вызвать путаницу в данных, отображаемых у пользователя. Проверьте, в каких часовых поясах работают ваши серверы, и убедитесь, что код выполняется корректно. Тестирование запланированных задач часто упускают из виду, особенно если они выполняются по расписанию, а не сразу после события.

По возможности используйте UTC во всех компонентах приложения: в базах данных, API, запланированных задачах и т. д. Это обеспечит согласованность и предотвратит ошибки, связанные с локальным временем сервера.

Также стоит провести ревизию конфигурационных файлов. Проверьте `tsconfig`-файлы, чтобы убедиться в оптимальных настройках для продакшена. В сборке

могли остаться файлы для стейджинг-сред, например мок-данные или переключатели для принудительного изменения представлений. В продакшен-среде они не нужны, а главное, подобный функционал для разработчиков не должен быть доступен пользователям. Эти элементы не только увеличивают размер артефактов сборки, но и могут создавать уязвимости.

Ключевая задача — попытаться сломать работу приложения всеми возможными способами до того, как это сделают пользователи. Поэтому полезно проводить тестирование в продакшене сразу после развертывания, проверяя работоспособность изменений как во фронтенде, так и в бэкенде. Однако такое тестирование нужно тщательно планировать, чтобы не мешать реальным пользователям. Например, можно использовать тестовые аккаунты и изолированные фейковые данные. Обязательно предупреждайте всех коллег о проведении тестов в продакшене, чтобы странное поведение системы не было ошибочно принято за реальные проблемы. Бывает очень неловко признавать, что даже после тщательного тестирования и усердной работы команды над качественным кодом могут оставаться баги.

Проблемы со сторонними сервисами

Не все проблемы связаны с вашим кодом, поскольку вы будете работать со сторонними сервисами и пакетами, код которых поддерживается другими организациями. Когда вы переключаетесь на учетные данные продакшен-среды и отключаете тестовый режим в стороннем сервисе, поведение приложения может измениться. Именно в этот момент необходимо убедиться, что сервис настроен корректно.

У меня был случай работы со Stripe, когда мы потратили столько времени на тестирование в препродакшене, что к моменту релиза в Stripe не было настроено ни одного продукта в продакшен-режиме. Ошибки, которые мы получали в продакшене, отличались от тех, с которыми мы проводили тесты, и мы даже столкнулись с событиями, которых не ожидали. Потребовалось два месяца непрерывного тестирования в продакшене, прежде чем мы убедились, что можно уведомлять пользователей о готовности новой функции. Совместные усилия фулстек-разработчиков, команд QA и DevOps, а также продуктовой команды позволили довести функционал до состояния, когда все пограничные случаи обработаны, а документация позволяет уверенно поддерживать пользователей.

Даже если функция уже в продакшене, пользователям не обязательно знать о ней сразу. Вы можете скрыть ее за фича-флагом, как мы обсуждали в главе 25, предоставив доступ только пользователям с определенными правами. Фича-флаг функции — это механизм включения/отключения конкретной возможности. Такие флаги часто используют на фронтенде, чтобы скрыть функциональность от пользователей, пока бэкенд не обновлен или не завершено дополнительное тестирование. Это дает возможность тестировать в продакшене, не затрагивая текущих пользователей. Не забудьте отключить фича-флаг, когда будете готовы показать функцию всем пользователям! Если этого не сделать, могут возникнуть проблемы с ожидаемыми сроками релиза.

Сторонние сервисы добавляют в ваше приложение новый тип неопределенности. Один из способов не отставать от изменений — регулярно проверять сайты используемых внешних сервисов на предмет анонсов предстоящих обновлений. Обращайте внимание на письма от этих сервисов о внесенных изменениях или новых функциях, над которыми они работают. Раз в месяц просматривайте документацию внешних сервисов, чтобы не пропустить «тихие» релизы бета- или альфа-версий. Это поможет вам заранее подготовиться к потенциальным проблемам, которые могут нарушить работу вашего приложения.

Помните, что не все критические изменения зависят от обновлений версий пакетов, установленных на фронтенде или бэкенде. Иногда сервисы просто меняют ответы или события независимо от используемой вами версии. Именно поэтому так важны ваши логи и оповещения — они помогут отследить эти неожиданные изменения. Одни сторонние сервисы вовремя уведомляют о предстоящих обновлениях, другие — нет.

Если вы анализируете логи и обнаруживаете ошибку, связанную с внешним сервисом, убедитесь, что сообщение об ошибке не подавляется каким-либо логом, который вы добавили в код. Вам нужно «сырое» сообщение об ошибке от сервиса, чтобы определить, на какой стороне проблема: на их или на вашей. Это особенно важно, когда сторонний сервис требует действий от пользователя, например предоставления вашему приложению доступа к его аккаунту в этом сервисе. Google Analytics (https://oreil.ly/nq3sj) — хороший пример такого сервиса.

В вашем приложении могут возникать ошибки из-за того, что пользователь не предоставил ему доступ к своим данным. С этой проблемой я сталкивалась в проектах: однажды три разработчика потратили целый день, чтобы понять ее причину, потому что в бэкенде все ошибки были переопределены кастомными сообщениями. Также возможны ситуации, когда ваше приложение использует взаимозависимые сервисы. Если один сервис не возвращает данные, другой начинает работать некорректно. В итоге приходится разбираться с комплексными проблемами, и могут уйти дни на временные решения, не говоря уж о долговременных.

Даже облачные сервисы периодически обновляются, что иногда приводит к отказу приложения. Ваша задача — отслеживать эти изменения и заранее продумывать реакцию, потому что пользователям не важно, почему приложение не работает. Главное, сообщить им, что вы в курсе проблемы и активно ее решаете. Можно скоординироваться с отделом поддержки или продаж, чтобы отправить пользователям уведомление о запланированном техническом обслуживании, если предстоит работа с проблемами в продакшене.

Тестирование сервисов в непродакшен-средах может быть затруднено из-за их ограничений. Настройка тестовых аккаунтов или проверка новых функций с использованием этих сервисов часто вызывает сложности, особенно если требуются реальные пользовательские данные. Учитывайте это при подготовке к выкладке

в продакшен. Именно поэтому так важно тестировать функцию непосредственно в продакшене, *прежде чем* объявлять ее готовой для пользователей.

Регулярное плановое обслуживание внешних сервисов помогает предотвратить множество проблем, но продакшен всегда может преподнести неожиданности. Ваша задача — минимизировать влияние сторонних сервисов и продумывать запасные варианты, даже если в какой-то момент придется разрабатывать собственное решение. Это искусство баланса между гибкостью, контролем, затратами и временем. Некоторые вещи невозможно реализовать без внешних сервисов, например получение специфичных пользовательских данных. Еще один компромисс, с которым сталкиваешься в продакшене: сторонние сервисы могут создавать для ваших пользователей уязвимости и подвергать атакам, которые вы не в силах контролировать.

Данные и безопасность в продакшене

Поскольку атаки в продакшене могут происходить из любого места и в любое время, необходимо тесно сотрудничать с командой ИБ, чтобы понять их подход к тестированию. Важно вовлечь продуктовую команду, команду дизайнеров и других ключевых стейкхолдеров в тестирование в продакшене. Чем больше людей взаимодействует с приложением, тем лучше организация в целом понимает его работу. Знание того, как приложение функционирует на всех уровнях стека, становится бесценным при обсуждении дорожной карты и новых функций.

Сосредоточьтесь на тестировании уязвимостей безопасности или лазеек в бизнес-логике. Если ранее не удалось внедрить инструменты для тестирования безопасности (например, упомянутые в главе 8, https://oreil.ly/1LuB4), сейчас самое время это сделать. Поскольку все компоненты уже подключены, можно точно определить, к каким данным имеет доступ каждый пользователь. Проверьте различные уровни доступа пользователей на фронтенде, чтобы убедиться, что полученный токен не предоставляет несанкционированный доступ к ресурсам бэкенда. Убедитесь, что сроки действия токенов истекают в соответствии с настройками. Попробуйте нарушить бизнес-правила, которые должны быть реализованы, и проверьте, удастся ли это сделать.

Например, в созданном вами приложении пользователь не должен иметь возможности заказать товар, которого нет в наличии. Но можно ли найти способ обойти это правило? Вы ограничили количество отображаемых товаров, но есть ли возможность сломать этот фильтр и увидеть все продукты? Именно такие лазейки в бизнес-логике вам нужно выявлять. Они могут становиться сложнее, если начать думать о способах манипуляции приложением через URL, локальное хранилище и хранилище сессий.

Вам следует проверить, существуют ли обходные пути для логики бэкенда через пользовательский интерфейс. Вы проверяете и валидируете строки как на фронтенде, так и на бэкенде? Это особенно критично на бэкенде, поскольку он защищает

вашу базу данных. В формах попробуйте отправлять SQL-запросы в виде строковых значений, а затем проверьте бэкенд, чтобы увидеть его реакцию. Это один из способов протестировать уязвимость вашего приложения к SQL-инъекциям (https://oreil.ly/yXtsX). В целом проверяйте, можно ли передать значения, которые не соответствуют ожидаемым на бэкенде.

Когда пользователь проходит аутентификацию в приложении, могут потребоваться дополнительные вызовы авторизации (https://oreil.ly/ZipN6) для получения расширенного доступа (*step-up*). Если такие вызовы есть, пользователь может получить информацию из этого процесса. Например, строки верификатора кода (code verifier strings) и закодированные строки разрешений пользователя иногда временно добавляются в URL этих вызовов авторизации. Сами по себе такие данные обычно бесполезны, но в сочетании с информацией, хранящейся в браузере, они обретают ценность.

Также необходимо дважды проверить срок действия токенов и убедиться, что база данных обновляет их статус. Кроме того, важно выяснить, могут ли пользователи манипулировать токенами в API-запросах, чтобы получить дополнительные разрешения. Любой может декодировать ваши JWT-токены, просмотреть их содержимое и изменить значения. Убедитесь, что измененные токены нельзя использовать для ручного вызова эндпоинтов через инструменты вроде Postman.

Это также подходящий момент, чтобы проверить ответы, которые приходят в браузер, и убедиться, что вы не раскрываете конфиденциальную информацию. Не стоит передавать в открытом виде такие данные, как адреса пользователей или другие чувствительные сведения. Они должны обрабатываться через JWT-токены или просто с помощью включенного HTTPS на сервере. Некоторые поля ввода, например пароли или номера социального страхования, — следует маскировать, чтобы окружающие не могли легко увидеть вводимые данные. Элемент ввода пароля (https://oreil.ly/S5rDr) делает это по умолчанию, а для других полей можно использовать пакеты вроде @react-input/mask (https://oreil.ly/2mdhT) и react-input-mask (https://oreil.ly/dEFo).

Контейнеризация вашего приложения

Даже во время разработки может возникнуть ситуация, когда приложение работает в одних окружениях (например, на вашей машине), но не запускается в других. Или вы можете заметить, что приложение ведет себя по-разному на препродакшен- и продакшен-серверах. Обычно это происходит из-за небольших различий в конфигурациях этих систем. Например, локально у вас может быть Node 21.2.0, а на продакшен-сервере — Node 18.19.1.

Подобные различия могут серьезно повлиять на функциональность приложения. Именно поэтому запуск приложения в контейнере — это подход, который используют многие команды. *Контейнеры* (https://oreil.ly/kWXOn) позволяют объединить все зависимости, необходимые для работы приложения, включая требуемую версию

среды выполнения и устанавливаемые библиотеки. Контейнеризация очень похожа на виртуализацию. Ключевое отличие в том, что контейнеры не эмулируют оборудование, как это делают виртуальные машины. Контейнеры просто работают изолированно на существующем оборудовании, но вы не можете, например, устанавливать драйверы устройств или взаимодействовать с USB-устройствами. Таким образом, контейнер — это как отдельный мир для приложения на сервере.

Наиболее популярный инструмент для этого — Docker (https://oreil.ly/wSBYI), хотя существуют и другие варианты, такие как Podman (https://oreil.ly/BOOFB), Buildah (https://oreil.ly/LtDmW) или runc (https://oreil.ly/WbC7O). Если вы настроите контейнеры в начале проекта, у всех членов команды разработчиков будет единое видение процесса. Мы рассмотрим процесс настройки контейнеров для фронтенд- и бэкенд-приложений с использованием Docker.

Для начала установите Docker Engine (https://oreil.ly/_ClV-) локально, чтобы запускать его на своей машине. Это поможет создать единообразную среду разработки для всей команды. Уметь помогать коллегам устранять препятствия для работы — важный навык. Вот почему базовые знания взаимодействия с этим инструментом будут полезны. От вас не требуется быть экспертом по контейнерам, но если вы вдруг обнаружите, что это интересное направление, — углубляйтесь.

Теперь вам потребуется внести изменения в файл `vite.config.ts`. Параметры в этом файле и файлах `tsconfig` напрямую влияют на итоговый артефакт сборки для продакшена. Поэтому если возникнут проблемы со сборкой, в первую очередь проверьте конфигурационные файлы. Вот необходимые изменения для `vite.config.ts`:

```
…
export default defineConfig({
  base: "/",
…
  preview: {
    port: 8080,
    strictPort: true,
  },
  server: {
    port: 8080,
    strictPort: true,
    host: true,
    origin: "http://0.0.0.0:8080",
    watch: {
      usePolling: true,
    },
  },
})
```

Мы начнем с создания образа для фронтенд-приложения. Образ — это шаблон, содержащий инструкции для создания контейнера со всеми библиотеками и зависимостями, необходимыми для работы приложения. Вы создаете образы, написав

Dockerfile (https://oreil.ly/XlTfg), который содержит команды Docker для сборки контейнера, где будет запущено приложение. Эти команды аналогичны тем, которые вы бы выполнили в терминале на сервере для настройки контейнера. Вот простой Dockerfile для запуска продакшен-сборки приложения в контейнере:

```
FROM node:21-alpine

WORKDIR /dashboard-ui

COPY package.json .

RUN npm install

COPY . .

RUN npm run build

EXPOSE 8080

CMD [ "npm", "run", "preview" ]
```

В этом Dockerfile используются шесть инструкций с аргументами.

- Инструкция `FROM` (https://oreil.ly/X66as) задает базовый образ (https://hub.docker.com), на основе которого строится контейнер. Аргументом обычно указывается версия среды выполнения.
- Инструкция `WORKDIR` (https://oreil.ly/tKFaO) устанавливает рабочую директорию в контейнере для других инструкций Dockerfile. Это позволяет задать расположение, куда будут скопированы файлы вашего приложения в контейнере.
- Инструкция `COPY` (https://oreil.ly/Cy3Ml) часто используется в Dockerfile. Она позволяет переносить файлы из локальной директории компьютера в контейнер.
- Инструкция `RUN` (https://oreil.ly/kL80q) выполняет команды, которые обычно запускаются в терминале внутри контейнера.
- Инструкция `EXPOSE` (https://oreil.ly/aMQuT) сообщает Docker, как открыть указанный порт контейнера для хост-сервера. Этот процесс аналогичен ранее упомянутой виртуализации: контейнер остается изолированным, но появляется способ взаимодействия с ним для хост-сервера.
- Наконец, инструкция `CMD` (https://oreil.ly/qk_KW) задает команду, которая будет выполняться при запуске контейнера из образа.

Чтобы создать образ из этого Dockerfile, выполните следующую команду в терминале директории вашего фронтенд-приложения:

```
docker build . -t "dashboard-ui:v1.0"
```

Эта команда собирает образ, определенный в Dockerfile. Точка между `build` и `-t` крайне важна, но ее легко пропустить, — она указывает, что команда должна

выполняться в текущей директории. Опция `-t` задает имя и тег для образа (в данном случае `dashboard-ui:v1.0`). Хотя можно использовать автоматически сгенерированный ID, именование полезно при работе с большим количеством образов.

Запуск в контейнере только фронтендовой части во время разработки

В данном примере используется команда `npm run preview`, поскольку контейнер запускается локально. Документация Vite (https://oreil.ly/FK1Ff) не рекомендует использовать этот подход в продакшене, поэтому он подходит только для локальной разработки:

```
FROM node:21-alpine

WORKDIR /dashboard-ui

COPY package.json .

RUN npm install
RUN npm install serve -g

COPY . .

RUN npm run build

EXPOSE 8080

CMD [ "serve", "-s", "dist" ]
```

Для реального продакшен-фронтенда приложение следует развернуть в S3-бакете, Vercel, Cloudflare или аналогичном сервисе.

После выполнения команды у вас будет образ, который послужит основой для контейнера. Для просмотра текущих образов Docker используйте:

```
docker images
```

Вывод будет выглядеть так:

```
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
dashboard-ui  v1.0      8179b8302716   21 seconds ago  1.43GB
```

Наконец, для запуска контейнера на основе этого образа выполните:

```
docker run -d -p 8080:8080 dashboard-ui:v1.0
```

Опция `-d` позволяет запускать контейнер в фоновом режиме (detached mode): в этом случае контейнер работает в фоне, а не отображает процесс в вашем терминале. Опция `-p` отображает порта хоста (порта на сервере) на порт контейнера (порт

внутри контейнера), чтобы приложение было доступно и работало на указанном в Dockerfile порту. Теперь вы сможете получить доступ к вашему приложению через контейнер по адресу `localhost:8080`.

Вы также можете увидеть запущенный контейнер в терминале с помощью следующей команды и вывода:

```
docker ps
```

Эта команда аналогична команде `ps` (process) в Unix-системах (https://oreil.ly/wc-WAW). Таким образом вы увидите текущие запущенные процессы в контейнере, как показано на рис. 26.1.

Рис. 26.1. Докер-контейнер в работе

У вас будет аналогичный Dockerfile для бэкенда. Основное отличие от фронтенда заключается в аргументе инструкции `CMD` для запуска приложения. Вот как это будет выглядеть:

```
FROM node:21-alpine

WORKDIR /dashboard-server

COPY package.json .
COPY package-lock.json .
RUN npm i --production

COPY src src
RUN npm run build

EXPOSE 3000

CMD ["node", "dist/main.js"]
```

При создании новых образов вам следует учитывать концепцию золотых образов (golden images). *Золотой образ* — это шаблон с предопределенным окружением, инструментами, пакетами и настройками безопасности для всех создаваемых вами образов. Эти образы полезны по нескольким причинам:

- они регулярно пересобираются для поддержания актуальных патчей безопасности;
- они ускоряют разработку приложений, избавляя вас и команду от необходимости настраивать окружение;
- они обеспечивают согласованность между всеми образами;
- они позволяют автоматизировать развертывание новых приложений.

Если в вашей организации есть выделенная команда DevOps, именно она обычно отвечает за поддержание золотого образа в актуальном состоянии.

Заключение

В этой главе мы рассмотрели некоторые проблемы интеграции, возникающие в продакшене, и методы их решения. Наиболее непредсказуемые проблемы часто связаны со сторонними сервисами. Поскольку вы не контролируете их изменения, важно регулярно мониторить их документацию и анонсы, а также проводить тестирование в продакшене, чтобы быть в курсе изменений. Многие проблемы интеграции можно решить за счет координации между разработчиками и другими командами.

Большая часть главы посвящена настройке Docker и созданию контейнеров для обеспечения согласованности развертываний в продакшене. Вам следует сотрудничать с командой DevOps, чтобы поддерживать файлы Dockerfile в актуальном состоянии с учетом требований безопасности и приложения, — об этом мы поговорим в следующей главе. Понимание основ Docker поможет вам поддерживать образы для ваших приложений. Это позволит разработчикам не зависеть от команды DevOps при внесении небольших обновлений версий или изменений скриптов.

ГЛАВА 27

Построение CI/CD-пайплайна

Теперь, когда вы прошли процесс интеграции фронтенда и бэкенда вместе со всей необходимой инфраструктурой и сервисами, вы можете автоматизировать свои развертывания с помощью CI/CD-пайплайна. Такая автоматизация гарантирует, что все ваши развертывания в продакшен или другие среды будут выполняться одинаково каждый раз. Такой подход стал стандартом практически для всех организаций.

Автоматизированный CI/CD-пайплайн побуждает команду разработчиков делать небольшие и частые релизы. Вам нужно будет работать в связке с командой DevOps для создания и поддержки этого пайплайна. Вы сможете внести свой вклад, если будете хорошо понимать, как работает приложение и как оно должно выполняться. Это поможет вам устанавливать переменные окружения и версии среды выполнения (runtime), а также убедиться, что выполняются правильные команды для подготовки приложения к развертыванию.

В этой главе мы рассмотрим:

- как использовать CircleCI для настройки CI/CD-пайплайна;
- как выбирать инструменты для упрощения пайплайнов;
- как настраивать различные среды и их назначение;
- как использовать GitHub в рамках этого процесса.

Существует множество вариантов и инструментов для настройки пайплайна, и я рассмотрю те, которые помогут вам быстро войти в курс дела. Это область, которая тесно пересекается с работой команды DevOps, поэтому от вас не требуется быть здесь экспертом. Взаимодействуйте с ней, чтобы получить полезную информацию для стабильной работы своей команды, а затем решите, нужно ли углубляться дальше. После создания собственного пайплайна вы сможете модифицировать его по мере изменения потребностей проекта.

Создание собственного пайплайна

Развертывание приложения в любой среде предполагает выполнение ряда команд в терминале на сервере. Такое развертывание включает в себя шаги по:

- загрузке всех пакетов;
- сборке приложения или контейнера;
- настройке переменных окружения;
- обработке дополнительной конфигурации;
- запуску приложения на сервере.

Для гладкого развертывания может потребоваться множество шагов. Именно здесь используется автоматизация — она исключает риск пропуска шагов или их выполнения в неправильном порядке, если делать это вручную.

Ваш CI/CD-пайплайн обеспечивает автоматизацию и, следовательно, согласованность всего процесса. Пайплайн состоит из шагов и этапов:

- *шаг* — отдельная команда для выполнения задачи (например, создание сборки или инвалидация кэша);
- *этап* — группа шагов для достижения конкретной цели (например, тестирование или развертывание артефакта в сервис).

Минимальный набор включает этап сборки, тестирования и развертывания. Могут быть добавлены дополнительные этапы (см. https://oreil.ly/lds1Q). На *этапе сборки* выполняются шаги для получения актуальной версии кода из репозитория команды. Здесь объединяются все индивидуальные функции и исправления, над которыми вы с командой работали.

Типичные шаги на этапе сборки включают:

- подготовку среды;
- настройку переменных окружения;
- получение кода из репозитория;
- установку всех пакетов;
- запуск команд сборки.

Этап тестирования — это когда вы запускаете модульные тесты, интеграционные тесты и, возможно, некоторые тесты безопасности для версии приложения, готовой к развертыванию. В условиях цейтнота команды могут испытывать соблазн пропустить этот этап, поскольку выполнение тестов занимает определенное время (в зависимости от используемых инструментов). Однако не следует поддаваться этому соблазну ради более быстрого развертывания. Ваши тесты нужны для того, чтобы защитить пользователей от регрессионных ошибок, багов и уязвимостей.

Типичные шаги на этапе тестирования включают в себя:

- запуск тестов качества кода;
- выполнение модульных тестов;

- проведение интеграционных тестов;
- тестирование безопасности;
- отправку результатов тестов разработчикам.

Финальный этап — это *этап развертывания*, когда вы определяете, в какую среду будут выпущены изменения. На этом этапе можно также управлять другими процессами, например откатами, а также настроить его выполнение по расписанию или триггеру на основе определенных событий. Этот этап сильно варьируется в разных организациях, поскольку используемая стратегия зависит от отрасли, размера команд и договоренностей между инженерными и бизнес-подразделениями. Некоторые компании никогда не автоматизируют развертывание в продакшен, тогда как другие ожидают, что это должно происходить автоматически при выполнении определенных условий в пайплайне.

Типичные шаги на этапе развертывания включают:

- настройку секретов или подключений к облачному сервису, который будет обслуживать артефакт сборки;
- загрузку артефакта в облачный сервис;
- перемещение артефакта сборки на сервер;
- сброс кэш-сервиса для приложения;
- отправку оповещения об успешном или неудачном завершении развертывания.

Хотя конкретные шаги на каждом этапе зависят от организации и используемых инструментов, перечисленные действия можно реализовать, если вам потребуется настроить собственный пайплайн.

Вопросы производительности

При проектировании этих этапов важно, чтобы они работали быстро, но без ущерба для качества. Я сталкивалась с CI/CD-пайплайнами, выполнение которых занимало более часа. Это существенно замедляет работу команды, вызывает раздражение у всех участников и не позволяет выпускать небольшие инкрементные изменения. Каждый из этих этапов можно разбить на более мелкие подэтапы, а некоторые из них — выполнять параллельно, чтобы ускорить пайплайн.

Следите за временем выполнения при настройке пайплайна. Первое место, где можно получить мгновенную отдачу, — это пул-реквесты. Хороший пайплайн должен выявлять ошибки как можно раньше. Так разработчики смогут быстро исправлять проблемы, и вам не придется тратить время на ожидание завершения скриптов, чтобы в последний момент обнаружить ошибку в коде.

Одна из эффективных стратегий — запускать этапы сборки и тестирования для пул-реквестов, направленных в определенные ветки. Например, можно открыть черновик пул-реквеста до того, как он будет готов к ревью. На этом этапе полезно

проверить, проходят ли тесты и успешно ли завершается сборка, чтобы исправить проблемы еще до ревью. При таком подходе можно получить отдачу на ранних стадиях разработки и избежать задержек с ревью и релизами в будущем. Еще более раннюю отдачу можно получить, если настроить некоторые из этих проверок с помощью git-хуков.

Одна из целей автоматизации развертывания — убедиться, что код будет стабильно работать в любой среде. Именно поэтому важно тщательно продумывать шаги для каждого этапа и документировать их. Лучшее, что можно сделать, — поддерживать актуальную документацию по таким процессам. Так вы сможете получать обратную связь, проверять свои идеи и принимать обоснованные решения, опираясь на коллективный опыт. Документирование CI/CD-пайплайна также помогает глубже проработать процесс, поэтому к моменту написания кода и настройки инструментов у вас уже будет четкое понимание общей картины.

Теперь, когда вы знаете, из чего состоит ваш пайплайн, можно начинать добавлять первые автоматизированные проверки.

Git-хуки

Поскольку ваш проект использует Git, вы можете настроить множество проверок, которые помогут выявлять проблемы задолго до развертывания. В вашем фронтенд-репозитории уже есть несколько git-хуков (https://oreil.ly/OMoGl) для действий `pre-commit` и `pre-push` с использованием Husky (https://oreil.ly/JZ2CO). Текущие команды обеспечивают контроль качества кода до того, как разработчик выпустит изменения. Сейчас в репозитории эти команды указаны в хуке `pre-commit`:

```
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"

npm run lint
npm run format
```

Вы можете добавить сюда дополнительные проверки качества кода и соблюдения стандартов разработки, чтобы контролировать сообщения коммитов и конфликты слияния с целевой веткой и ограничивать коммиты в защищенные ветки на локальном уровне. Давайте добавим новый хук для проверки сообщений коммитов, которые пишут разработчики. Создайте в директории .husky вашего фронтенд-репозитория новый файл `commit-msg` и добавьте следующий скрипт для проверки формата сообщений:

```
#!/usr/bin/env bash
. "$(dirname -- "$0")/_/husky.sh"

commit_types="(build|chore|ci|docs|feat|fix|perf|refactor|revert|style|test|wip)"
conventional_commit_regex="^${commit_types}(\([a-z \-]+\))?!?: .+$"
```

```
commit_message=$(cat "$1")

if [[ "$commit_message" =~ $conventional_commit_regex ]]; then
  exit 0
fi

echo "Сообщение коммита не соответствует стандартам"
echo "Пример корректного сообщения:"
echo " feat: обновление полей модального окна"

exit 1
```

Такой скрипт упростит быстрый просмотр объединенных пул-реквестов и понимание внесенных изменений в общих ветках. Это особенно полезно при отладке проблем, возникающих в средах после развертывания. Вам и команде стоит договориться о формате сообщений, но как минимум они должны позволять быстро понять, что именно реализовано в каждом пул-реквесте. Можно использовать номера тикетов, классификацию изменений (например, исправления, баги, фичи) или другие типы, определенные в спецификации Conventional Commits (https://oreil.ly/3LWM-). Также можно рассмотреть инструменты на основе ИИ, такие как aicommit (https://oreil.ly/KduBa) или aicommits (https://oreil.ly/Fn2_G), для дальнейшей автоматизации этой задачи.

Еще один полезный git-хук — `pre-push`. Он запускает скрипт при попытке разработчика отправить изменения в удаленную ветку, даже если это его личная ветка. Сейчас в этом хуке настроен запуск модульных тестов перед отправкой кода. Вот текущий скрипт:

```
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"

npm test
```

Будьте осторожны с командами, которые выполняются в хуках `pre-commit` и `pre-push`, так как иногда они могут мешать коллегам делиться незавершенной работой. Это может привести к ситуации, когда разработчики просто закомментируют все проверки в `pre-push` и в общие ветки попадет код более низкого качества. Выбирайте команды для этих хуков осознанно, чтобы они оставались полезными. Помните, что у вас еще есть множество проверок, которые выполняются в CI/CD-пайплайне.

Хочу отметить, что вам не обязательно использовать такие инструменты, как Husky, для работы с git-хуками. Когда вы работаете с репозиторием, использующим Git, вы можете заглянуть в директорию .git/hooks проекта и найти там примеры всех возможных хуков (см. рис. 27.1). Учтите, что директория .git, скорее всего, скрыта в вашем файловом менеджере, поэтому вы можете не найти ее сразу. Некоторые разработчики предпочитают Husky встроенным хукам Git, потому что он автоматизирует большую часть настройки.

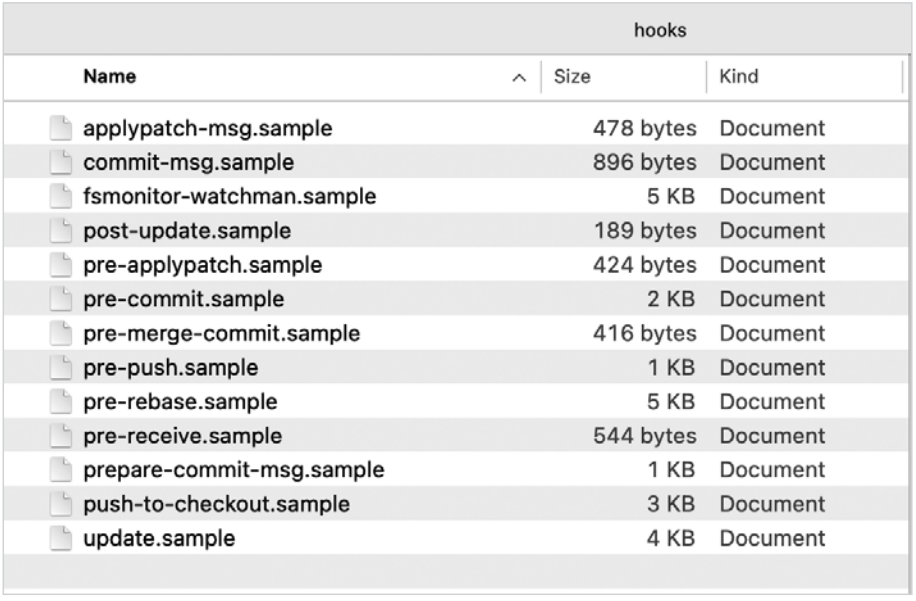

Рис. 27.1. Git-хуки в директории .git/hooks

Вот пример того, как можно реализовать хуки `pre-commit` и `pre-push` на чистом Git. Эти скрипты уже существуют — вам достаточно удалить расширение `.sample` из имени файла нужного хука.

Хук `pre-commit`:

```
#!/bin/sh
# Хук pre-commit
# Пример скрипта для проверки изменений перед коммитом.
# Вызывается командой "git commit" без аргументов. Хук должен
# завершиться с ненулевым статусом, если нужно отменить коммит,
# предварительно выдав поясняющее сообщение.
# Чтобы активировать этот хук, переименуйте файл в "pre-commit".

if git rev-parse --verify HEAD >/dev/null 2>&1
then
    against=HEAD
else
    # Первый коммит: сравниваем с пустым деревом
    against=$(git hash-object -t tree /dev/null)
fi

# Разрешить не-ASCII имена файлов (false по умолчанию)
allownonascii=$(git config --type=bool hooks.allownonascii)

# Перенаправляем вывод в stderr
exec 1>&2
```

```
# Кросс-платформенные проекты обычно избегают не-ASCII имен файлов;стараемся не
# добавлять их в репозиторий.
# Проверяем, что в именах файлов только печатные ASCII-символы (от пробела до #
тильды).
if [ "$allownonascii" != "true" ] &&
   # Обратите внимание, что квадратные скобки в диапазоне tr допустимы
   # (и даже необходимы для совместимости с /usr/bin/tr в Solaris 10),
   # так как сами символы скобок попадают в указанный диапазон.
   test $(git diff --cached --name-only --diff-filter=A -z $against |
      LC_ALL=C tr -d '[ -~]\0' | wc -c) != 0
then
      cat <<\EOF
```

Хук `pre-push`:

```
#!/bin/sh
# Хук pre-push
# Пример скрипта для проверки изменений перед отправкой в удаленный репозиторий.
Вызывается командой "git push"
# после проверки состояния удаленного репозитория, но до фактической отправки
изменений.
# Если скрипт завершится с ненулевым статусом, отправка изменений будет отменена.
# Хук получает следующие параметры:
# $1 — Имя удаленного репозитория, в который выполняется отправка
# $2 — URL удаленного репозитория
# При отправке без указания имени удаленного репозитория оба аргумента будут
одинаковыми.
# Информация о коммитах, которые отправляются, передается через стандартный ввод
в формате:
# <локальная ссылка> <локальный OID> <удаленная ссылка> <удаленный OID>
# Этот пример показывает, как запретить отправку коммитов, сообщения которых
начинаются с "WIP" (work in progress — работа в процессе).

remote="$1"
url="$2"

zero=$(git hash-object --stdin </dev/null | tr '[0-9a-f]' '0')

while read local_ref local_oid remote_ref remote_oid
do
    if test "$local_oid" = "$zero"
    then
        # Обработка удаления
        :
    else
        if test "$remote_oid" = "$zero"
        then
            # Новая ветка — проверяем все коммиты
            range="$local_oid"
        else
            # Обновление существующей ветки — проверяем только новые коммиты
            range="$remote_oid..$local_oid"
        fi
```

```
        # Проверяем наличие WIP-коммитов
        commit=$(git rev-list -n 1 --grep '^WIP' "$range")
        if test -n "$commit"
        then
            echo >&2 "Обнаружен WIP-коммит в $local_ref, отправка отменена"
            exit 1
        fi
    fi
done

exit 0
```

Теперь вы используете жизненный цикл Git для выполнения стандартных проверок до того, как код попадет в CI/CD-пайплайн. Это экономит значительное время на ревью пул-реквестов, запуск пайплайна и ожидание отчетов о проблемах, которые можно было выявить гораздо раньше. Вы можете добавить больше хуков, если вы с командой считаете, что это поможет сэкономить время или поддерживать единообразие.

Конфигурирование GitHub

Вы добавили инструменты для максимально раннего выявления проблем перед тем, как разработчик выпустит код. Теперь пришло время перейти к следующему уровню — пул-реквестам на GitHub. Здесь вы можете усилить контроль качества кода, а также выполнить первоначальное тестирование, чтобы убедиться, что код работает как задумано и не ломает очевидные вещи. Установленные в начале проекта соглашения отлично подойдут для этого этапа, так как вы уже предусмотрели, что должно проверяться во время ревью пул-реквестов.

В вашем репозитории на GitHub следует настроить несколько параметров, чтобы защитить ветки от недостаточно тщательно выполненных ревью. Рекомендуется запретить прямые пуши в ветки `main` и `develop`, так как именно от них обычно создают ветки разработчики и выполняют развертывание. Также стоит ограничить список веток, которые можно удалять, а также список лиц, имеющих права на такое удаление и другие возможные действия. Эти настройки можно найти в разделе настроек репозитория (пример на рис. 27.2).

Изучите доступные настройки GitHub — вы найдете подходящие варианты практически для любого правила, которое хотите внедрить. Подробнее о стратегиях работы с ветками Git я расскажу в главе 28, поскольку это существенно влияет на процессы разработки, развертывания и реагирования на инциденты. Пока сосредоточимся на базовой структуре:

- ветка `main` — для развертывания в продакшен;
- ветка `develop` — для стейджинга или препродакшена;
- ветки `dev` — для работы команды.

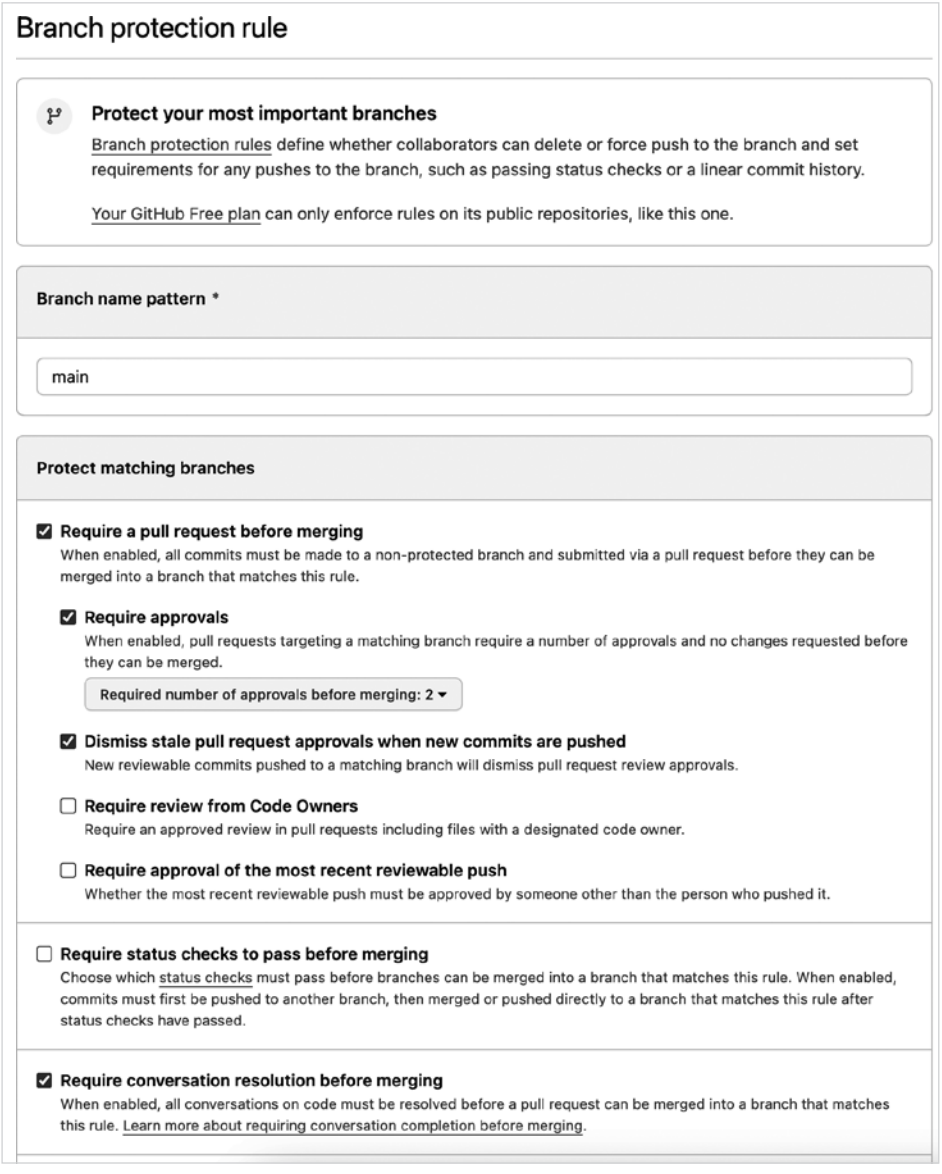

Рис. 27.2. Настройки веток в GitHub

Хорошее правило — требовать хотя бы одно подтверждение пул-реквеста перед мержем ветки разработчика в `develop`. Так вы и команда будете уверены, что код соответствует соглашениям, а улучшения можно обсудить и внедрить. Минимальная проверка в любом пул-реквесте может выглядеть так:

- запустить ветку `develop` локально;
- убедиться, что приложение работает и соответствует критериям приемки из тикета.

Еще одно правило для настройки GitHub — запретить прямые пул-реквесты и пуши в ветку `main`. Это ваш «источник правды» для кода, который работает в продакшене, и вам не нужно, чтобы кто-то мог напрямую изменять его, минуя процесс релиза. Однако учитывайте это при работе с хотфиксами: иногда может потребоваться временно отключить это правило, чтобы быстро внести исправление в продакшен.

Дополнительное правило — запускать часть проверок пайплайна прямо в пул-реквесте. Например, убедиться, что тесты проходят и сборка успешно создается, — это поможет выявить проблемы до слияния и избежать блокировки работы команды, пока разработчик разбирается с ошибками.

Такие правила и ограничения повышают уверенность, что код, написанный вами и командой, будет работать в продакшене. Кроме того, они помогают поддерживать чистоту кода, избегая файлов с разным форматированием или несогласованными стилями методов и переменных. Теперь, когда вы предусмотрели выявление проблем на этапах ревью и слияния, можно переходить к настройке пайплайна.

Конфигурирование CircleCI

Существует множество инструментов для CI/CD-пайплайнов, таких как Jenkins (https://jenkins.io), CircleCI (https://circleci.com), Bamboo (https://oreil.ly/IycNX), GitHub Actions (https://oreil.ly/uGb3g) и TeamCity (https://oreil.ly/YDmDC). Я буду использовать CircleCI для настройки пайплайна этого приложения, потому что он требует минимальной конфигурации и хорошо интегрируется с GitHub, Bitbucket и другими системами. Чтобы начать работу с CircleCI, зарегистрируйте бесплатный аккаунт (https://circleci.com/signup) и подключите его к репозиторию GitHub, для которого нужно настроить пайплайн. После регистрации рекомендую ознакомиться с кратким руководством по подключению репозитория и созданию начального конфигурационного файла CircleCI. После входа в систему и выбора репозитория GitHub для настройки нового пайплайна вы увидите интерфейс, как на рис. 27.3.

Я выбрала ветку `main` для настройки пайплайна и опцию Fastest. Конфигурационный файл пайплайна будет находиться в новой директории .circleci в корне проекта — это файл `config.yml`. Теперь, если зайти в репозиторий GitHub, вы увидите новую ветку `circleci-project-setup`, содержащую начальный конфигурационный файл CI/CD. Откройте пул-реквест для мержа этой ветки в `main`, и вы увидите новый файл, как показано на рис. 27.4.

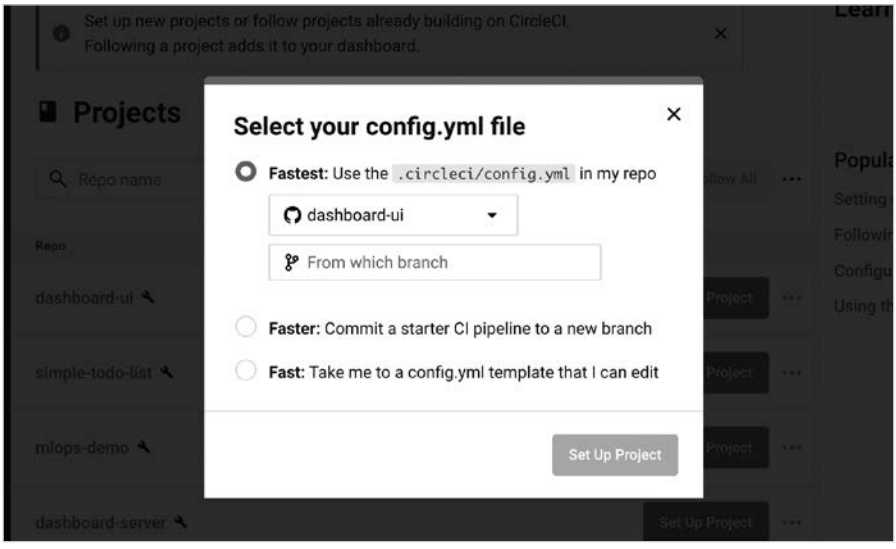

Рис. 27.3. Начальная настройка пайплайна в CircleCI

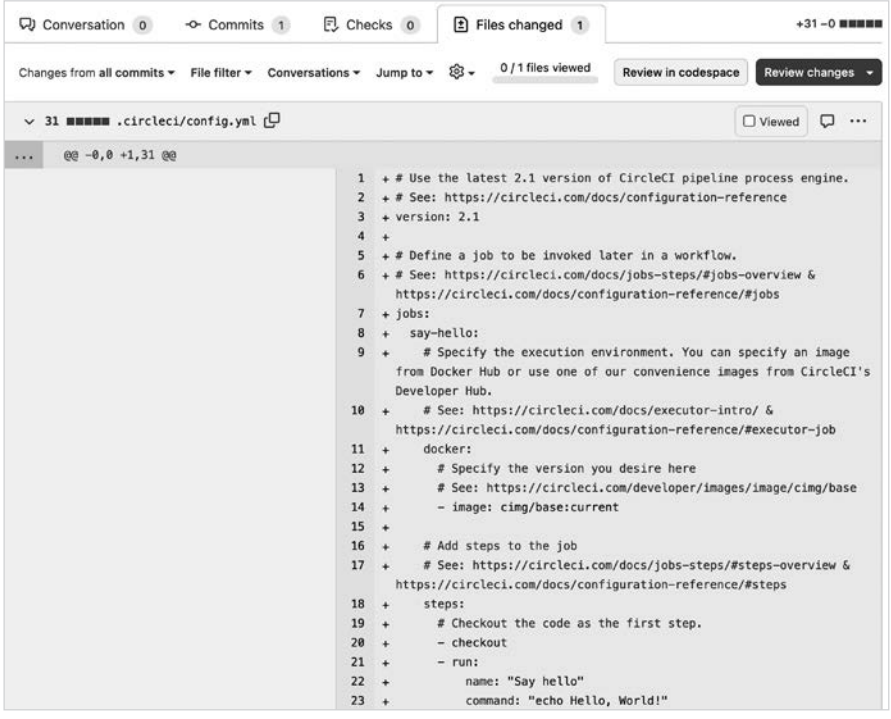

Рис. 27.4. Пул-реквест в GitHub для добавления CI/CD-конфигурации, сгенерированной CircleCI

В дальнейшем вы внесете множество изменений в этот файл, чтобы настроить собственные этапы и шаги пайплайна. Но сейчас важно убедиться, что пайплайн работает с вашего репозитория. Откройте пул-реквест с любыми изменениями — даже просто с обновлением README — и вы увидите примерно то, что показано на рис. 27.5.

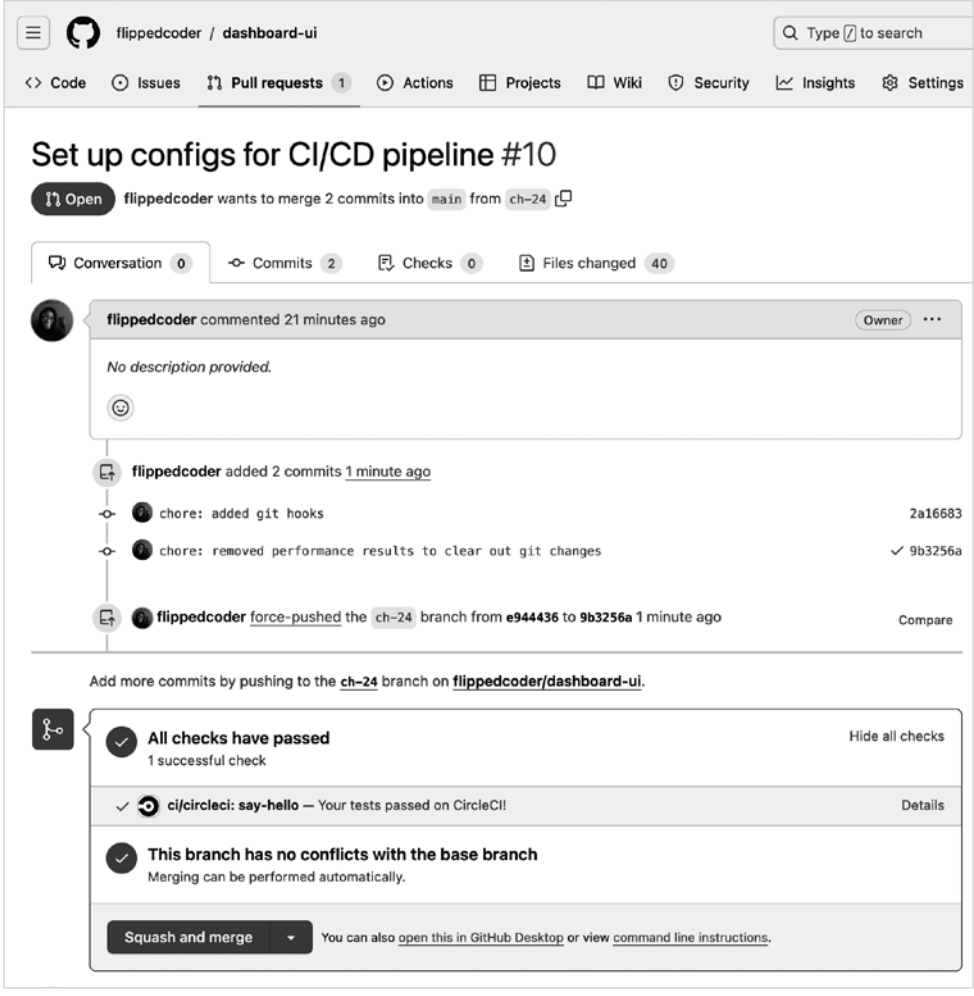

Рис. 27.5. Статус CircleCI в пул-реквесте GitHub

Теперь у вас есть проверка, которая будет запускаться для каждого пул-реквеста — как от вас лично, так и от команды. Чтобы узнать подробности о выполняемых проверках, перейдите в панель управления CircleCI. Там вы увидите интерфейс, аналогичный рис. 27.6.

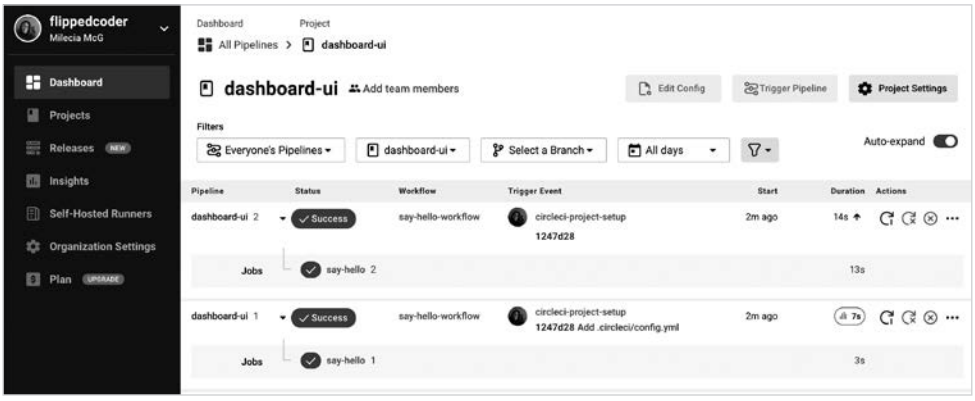

Рис. 27.6. Панель управления CircleCI

Отсюда следует перейти в workflow под названием `say-hello-workflow`, а затем выбрать задание (job) `say-hello` внутри него. Это отобразит шаги, которые были выполнены в рамках задания, как показано на рис. 27.7.

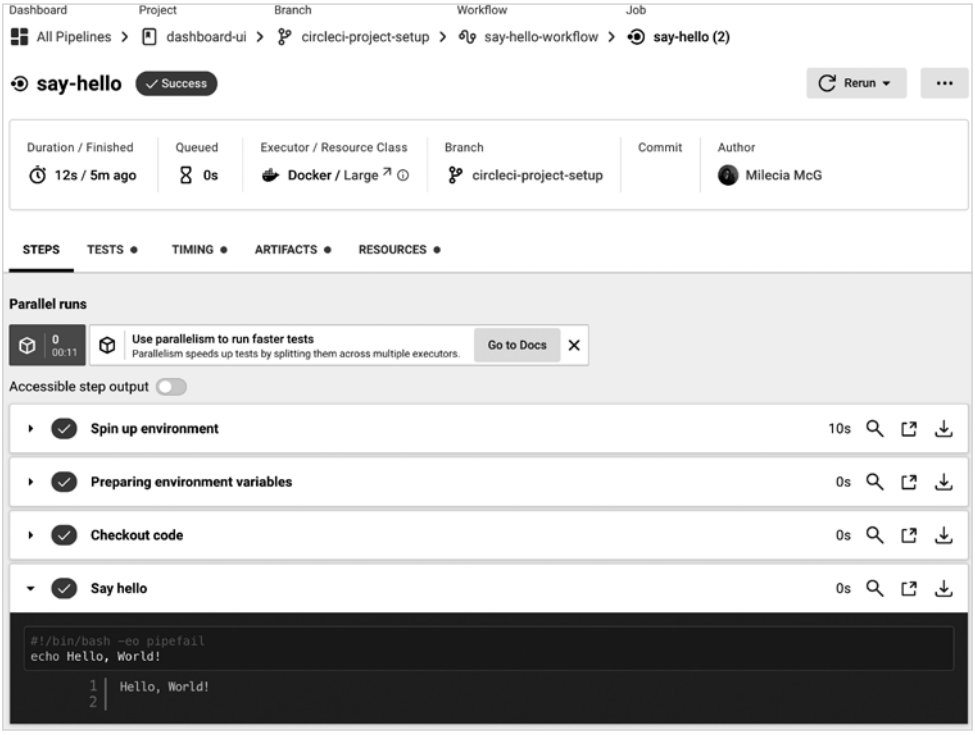

Рис. 27.7. Шаги задания say-hello

Теперь, когда вы подтвердили, что пайплайн подключен к вашему репозиторию, можно приступать к созданию собственных этапов и настройке пайплайна для развертывания кода в продакшен и другие среды. Рекомендуется делать это поэтапно, чтобы убедиться, что каждый шаг работает так, как ожидается. Это сэкономит время при отладке конфигурации пайплайна.

Начнем с этапов сборки и тестирования, которые мы обозначили ранее. Добавьте следующий код в ваш `config.yml`:

```
version: 2.1
orbs:
  node: circleci/node@5.1.1
  cypress: cypress-io/cypress@3
  snyk: snyk/snyk@2.1.0
jobs:
  unit-test:
    docker:
      - image: 'cimg/base:stable'
    steps:
      - checkout
      - node/install:
          node-version: '21.2'
      - node/install-packages
      - run:
          command: npm run lint
      - run:
          command: npm run test
  security-scan:
    docker:
      - image: 'cimg/base:stable'
    steps:
      - checkout
      - node/install:
          node-version: '21.2'
      - node/install-packages
      - snyk/scan
  integration-test:
    docker:
      - image: 'cimg/base:stable'
    steps:
      - checkout
      - node/install:
          node-version: '21.2'
      - node/install-packages
      - snyk/scan
  build:
    docker:
      - image: 'cimg/base:stable'
    steps:
      - checkout
      - node/install:
          node-version: '21.2'
```

```
      - node/install-packages
      - run:
          command: npm run build
workflows:
  build-and-tests:
    jobs:
      - build
      - unit-test:
          requires:
            - build
      - security-scan:
          requires:
            - unit-test
  integration-test:
    jobs:
      - cypress/run:
          start-command: npm run start
```

Это базовая настройка, и вам стоит поработать с командой DevOps, чтобы сделать ее более надежной. Однако и этого достаточно для создания пайплайна с несколькими проверками. Поскольку CircleCI — лишь один из возможных инструментов, я кратко объясню основные части этого файла, но для глубокого понимания стоит изучить концепции CircleCI (https://oreil.ly/pBwnc). Например, вы, скорее всего, будете запускать задачи последовательно в зависимости от результатов предыдущих, и в этом конфигурационном файле уже есть несколько таких примеров.

Первое, что бросается в глаза в конфигурации, — это `version`. Этот параметр определяет, какая версия CircleCI будет использоваться в пайплайне, а также какие ключи (https://oreil.ly/D_-0K) доступны для настройки. Далее идет раздел `orbs`. Орбы (https://oreil.ly/eIoZz) — это общие пакеты, содержащие задачи и команды, которые упрощают конфигурационный файл. Многие сторонние сервисы предоставляют орбы, так что вам не придется писать все команды с нуля. Именно так в этом пайплайне используются Cypress для интеграционного тестирования и Snyk для проверки безопасности без дополнительной настройки.

Затем следует раздел `jobs` — задачи. Задачи (https://oreil.ly/refei) содержат все шаги, которые должны выполняться на разных этапах пайплайна. Это один из самых детализированных разделов конфига, поскольку задачи — это строительные блоки для всего, что происходит в пайплайне. Наконец, есть раздел `workflows`. Воркфлоу (https://oreil.ly/YuOL3) определяют порядок выполнения задач в пайплайне на основе заданных требований.

Все эти разделы можно использовать более продвинутым способом, например, писать кастомные скрипты для задач и задавать более детальные требования в воркфлоу. В конечном счете можно прийти к воркфлоу и задачам для оркестрации контейнеров с помощью Kubernetes (https://kubernetes.io/docs), но это уже зона ответственности команды DevOps. Пока что, если вы отправите изменения конфига в репозиторий, в CircleCI вы увидите нечто подобное изображенному на рис. 27.8.

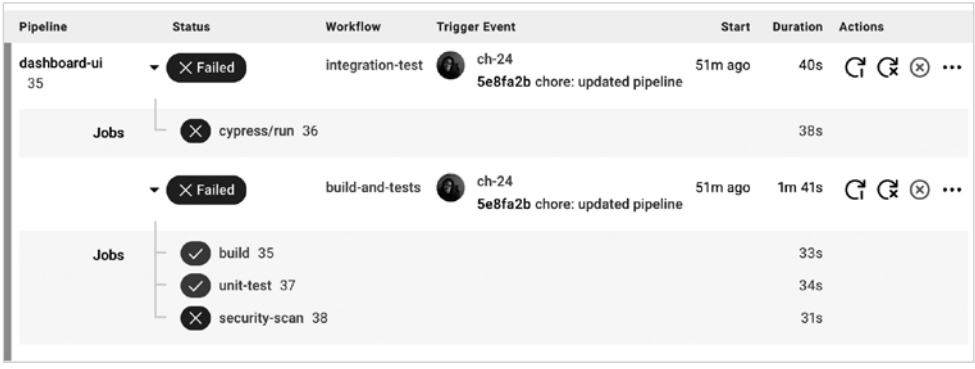

Рис. 27.8. CircleCI с новыми воркфлоу

Вы можете зайти в воркфлоу `build-and-tests`, чтобы увидеть детали выполняемых задач, как показано на рис. 27.9.

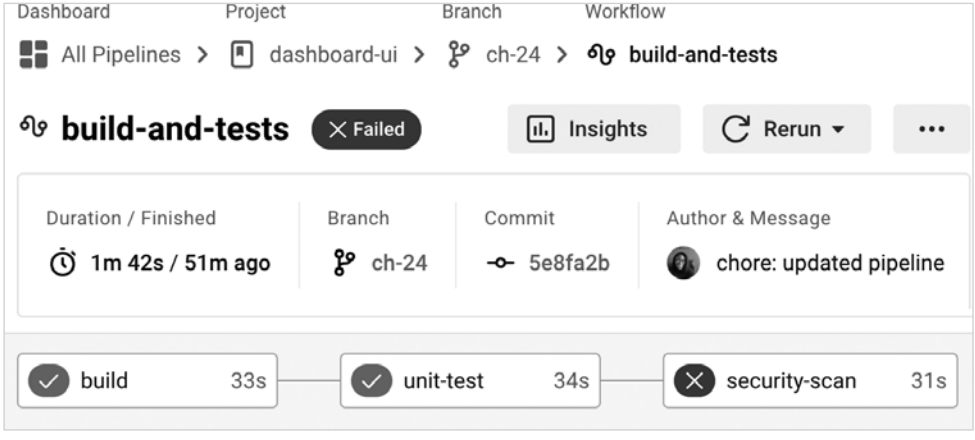

Рис. 27.9. Задачи в воркфлоу `build-and-tests`

Обратите внимание на время выполнения каждого воркфлоу и его задач — это поможет вам оценить общую продолжительность этапов пайплайна. Эти данные пригодятся при добавлении новых задач и шагов, а также для поиска возможностей оптимизации.

Теперь у вас настроен CI-пайплайн. Осталось только поработать с командой DevOps над этапом развертывания, где собранный артефакт передается в облачные сервисы для запуска на подготовленных серверах. Этот этап развертывания будет заключительным в контексте работы с командой DevOps, именно он обозначается аббревиатурой *CD* в CI/CD-пайплайне.

Пайплайны для разных сред

Последний этап настройки пайплайна — работа с несколькими средами для приложений. Как для бэкенд-, так и для фронтенд-приложений обычно используются среда разработки, стейджинг-среда и продакшен-среда. Для фронтенда могут быть дополнительные окружения (feature environment), где изменения тестируются изолированно перед переходом в следующую среду. Именно в такие дополнительные окружения (если они имеются) изменения попадают в первую очередь. Давайте рассмотрим эти окружения подробнее.

Дополнительное окружение

Дополнительное окружение — это место, где масштабные изменения во фронтенде проходят первичное тестирование перед слиянием с основной кодовой базой. Например, при рефакторинге крупного участка фронтенд-приложения, который существенно меняет представление или обработку данных, такое окружение помогает убедиться, что изменения ничего не ломают.

Дополнительные окружения также позволяют команде продолжать работу без блокировок: пока одни тестируют крупные изменения в изолированном окружении, другие могут интегрировать небольшие правки в окружение разработки и проверять их в общей системе.

Среда разработки

Среда разработки — это место, где команда разработчиков может выполнять предварительные развертывания, чтобы проверить, как их изменения функционируют в рамках всей системы. Развертывания здесь могут происходить в произвольном порядке, главное, согласовывать их внутри команды. В этой среде можно также привлекать команду QA для быстрого тестирования или демонстрации предстоящих изменений.

Фронтенд- и бэкенд-среды разработки обычно связаны между собой, чтобы можно было тестировать полный стек изменений. Часто дополнительные окружения для фронтенда подключаются к бэкенд-среде разработки. Это также хорошее место для проверки работы флагов функций, как обсуждалось в главе 25.

Стейджинг-среда

Стейджинг-среда используется для проверки корректности внесенных изменений. Здесь вы взаимодействуете с командой QA, которая проводит функциональное и регрессионное тестирование. Также можно предоставить доступ продуктовой команде и дизайнерам, чтобы они выполнили приемочное тестирование и внесли финальные правки. На этом этапе изменения для фронтенда и бэкенда должны быть синхронизированы, а приложение — подключено к сторонним сервисам для

проверки в условиях, максимально приближенных к продакшену. Здесь обычно используются тестовые учетные данные, которые могут быть доступны только в тестовой среде или также в среде разработки.

Продакшен-среда

Продакшен-среда — это среда, в которой пользователи взаимодействуют с приложением. После всех тестов в предыдущих средах вы должны четко понимать, чего ожидать с точки зрения пользователя. Обычно здесь используются наиболее мощные серверные ресурсы, а также обновляются переменные окружения для сторонних и облачных сервисов.

Переменные окружения

Переменные окружения позволяют управлять развертыванием в разных средах и настраивать подключения. Например, команда DevOps, скорее всего, развернет совершенно разную инфраструктуру для среды разработки и продакшена. Это включает отдельные базы данных, системы управления событиями и другие компоненты.

Существует несколько подходов к управлению переменными окружения в CI/CD-пайплайнах. В CircleCI можно задать переменные для каждого окружения в настройках проекта, а затем использовать их в файле `config.yml`. Теперь во всех частях приложения, где ссылаются на переменную окружения, будет подставляться заданное значение.

Ниже приведены новые задачи и воркфлоу в `config.yml`:

```
…
deploy-dev:
  docker:
    - image: 'cimg/base:stable'
  steps:
    - checkout
    - run:
        name: 'dev env vars'
        command: |
          echo $API_URL
deploy-staging:
  docker:
    - image: 'cimg/base:stable'
  steps:
    - checkout
    - run:
        name: 'staging env vars'
        command: |
          echo $API_URL
deploy-prod:
  docker:
    - image: 'cimg/base:stable'
```

```
  steps:
    - checkout
    - run:
        name: 'prod env vars'
        command: |
          echo $API_URL
…
deploy:
  jobs:
    - deploy-dev:
        filters:
          branches:
            only: develop
    - deploy-staging:
        filters:
          branches:
            only: staging
    - deploy-prod:
        requires:
          - build-and-tests
        filters:
          branches:
            only: main
```

На рис. 27.10 показано, как это будет выглядеть в интерфейсе CircleCI.

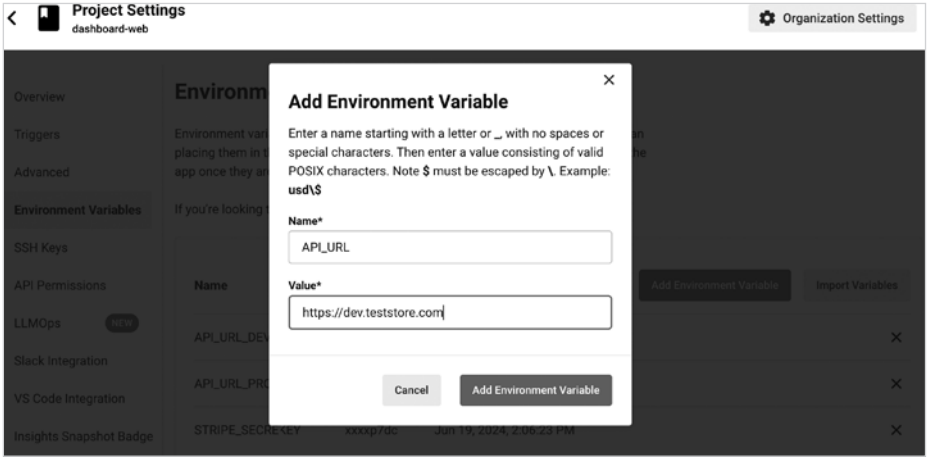

Рис. 27.10. Переменные окружения в CircleCI

Для выполнения этих новых задач потребуется тесное взаимодействие с командой DevOps, чтобы обеспечить корректное подключение к инфраструктуре. В примере обратите внимание, что приложение подключается к разным версиям переменной окружения `API_URL`, а процесс развертывания зависит от ветки, в которую были внесены изменения. Еще один важный момент: в этом воркфлоу этапы сборки

и тестирования должны быть успешно пройдены перед развертыванием в продакшен, а не в другие окружения. В примере эта проверка пропущена для наглядности, однако в реальном пайплайне она должна присутствовать. Обновленный пайплайн показан на рис. 27.11.

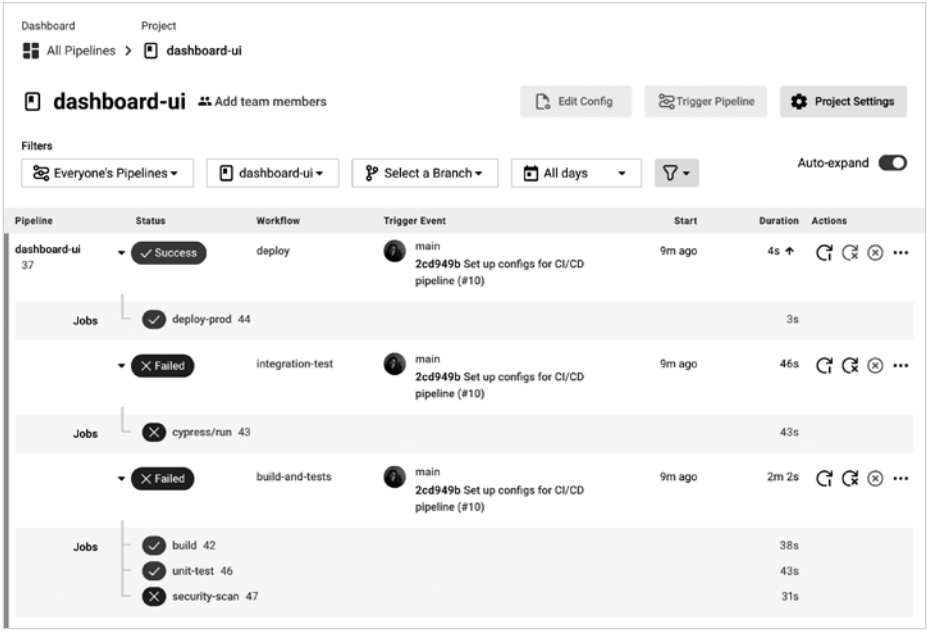

Рис. 27.11. Развертывание в продакшен в CircleCI

Альтернативный подход к работе с переменными окружения — использование отдельных `.env`-файлов для каждого окружения. Этот метод более рискованный: его можно применять в приватном репозитории или с помощью скрипта, который обновляет значения в памяти при прохождении приложения через пайплайн. В противном случае появляется риск, что все получат доступ к продакшен- и тестовым учетным данным. Важно помнить, что переменные окружения считаются лучшей практикой именно потому, что их значения не сохраняются где-либо, кроме облачного провайдера, который обеспечивает безопасную инфраструктуру для их хранения. В остальных случаях переменные загружаются в память и не сохраняются за ее пределами.

После настройки различных этапов развертывания в вашем пайплайне остается только согласовать с командой DevOps подключение ко всем необходимым сервисам и инфраструктуре, а затем проверить, что среды и окружения работают так, как ожидается.

Заключение

В этой главе мы разобрали создание собственного CI/CD-пайплайна. Вы узнали о различных этапах и задачах, которые необходимы для простого пайплайна. Рекомендую изучить пайплайны, используемые в вашей организации. Вам предстоит развертывать код в нескольких средах, и важно учитывать различия между ними при настройке пайплайнов.

Мы создали пайплайн с помощью CircleCI — это лишь один из множества инструментов для развертывания кода. Рекомендую ознакомиться и с другими решениями, чтобы понять, как они работают по сравнению с рассмотренным здесь вариантом. Если в вашем пайплайне есть все описанные этапы, вы можете совместно с командой DevOps добавить более сложную функциональность.

ГЛАВА 28

Управление Git

Владение Git — один из самых ценных навыков, который вы можете освоить. Чем бы вы ни занимались: работаете фулстек-разработчиком, специализируетесь на фронтенде или бэкенде или изучаете новый язык программирования, — вам необходимо уметь управлять изменениями в репозитории. Команда будет обращаться к вам за рекомендациями по работе с ветками и слиянию изменений в общих ветках.

Вы уже проделали большую работу по настройке процессов для команды: определили соглашения о коде, внедрили git-хуки для стандартизации коммитов, добавили правила для веток в GitHub и обеспечили соблюдение единого процесса ревью пул-реквестов. Теперь стоит углубиться в изучение различных команд и стратегий, которые помогут минимизировать конфликты, когда несколько разработчиков одновременно вносят изменения в удаленный репозиторий.

В этой главе мы поговорим о следующих аспектах:

- стратегия ветвления;
- управление слиянием с помощью команд Git;
- разрешение конфликтов при слиянии.

Одна из самых сложных задач — разрешить конфликты после слияния изменений, так как необходимо убедиться, что сохраняется правильная версия кода. Всегда полезно иметь несколько техник на случай конфликтов, особенно во время развертывания. Я расскажу о методах, которые обычно использую для управления ветками и слияниями изменений от разных команд, а также о других подходах, с которыми познакомилась за годы работы. Надеюсь, эти варианты помогут вам найти подходящее решение для ваших задач.

Стратегии ветвления

Существует несколько подходов к обращению с ветками при параллельной разработке изменений. Вот ключевые аспекты, которые стоит учитывать:

- пересекающиеся изменения в файлах;
- обновления версий пакетов;

- масштаб новой функции;
- планируемый срок выпуска новой функции;
- наличие изменений, затрагивающих все приложение.

Эти факторы помогут вам и команде определить оптимальный способ обработки входящих изменений. Часть стратегии ветвления будет зависеть от того, как настроены ваши пайплайны развертывания.

Стандартные ветки и процесс слияния

Как обсуждалось в главе 27, ваш код будет развертываться в разных средах. Эти среды обычно связаны с определенными ветками. Распространенная стратегия предполагает использование веток `main`, `staging` и `develop`. Названия могут отличаться в разных организациях, но их назначение будет схожим:

- ветка `main` — для развертывания в продакшен;
- ветка `staging` — для тестирования изменений перед выпуском в продакшен;
- ветка `develop` — здесь разработчики проводят внутреннюю проверку перед передачей в QA или другим командам.

Обе ветки — `staging` и `develop` — обычно связаны с соответствующими средами развертывания.

Когда пул-реквесты (https://oreil.ly/pmZvu) готовы к выпуску в продакшен, стандартный процесс заключается в интеграции пул-реквестов в ветку `develop`. После попадания изменений в `develop` команда разработчиков проводит первоначальные проверки. Как только `develop` проходит все проверки, изменения попадают в тестовую среду для утверждения командой QA и продуктовой командой. После успешной валидации в тестовой среде изменения мержатся в `main`, что запускает процесс развертывания в продакшен.

Важно помнить, что пул-реквесты — это специфика GitHub или Bitbucket, а не самого Git. В GitLab аналогичная концепция называется *merge request* (запрос на слияние). Вы можете мержить изменения через Git, вообще не создавая пул-реквесты.

Ветку `main` можно считать источником истины (source of truth) для приложения, поскольку именно ее видят пользователи. Если возникает вопрос о доступных функциях или корректности изменений, обычно следует обращаться именно к `main`, поскольку она напрямую влияет на пользователей. Ветка `staging` — это источник истины для всех изменений, готовых к релизу. Вероятно, она содержит меньше изменений, чем `develop`, но включает функциональность, которой еще нет в `main`. Ветка `develop` обычно отражает самые свежие функции, утвержденные командой разработчиков. Поэтому при начале работы над новой задачей именно с этой ветки стоит начинать. Пример этого процесса показан на рис. 28.1.

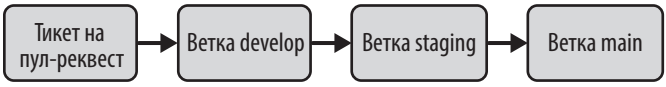

Рис. 28.1. Процесс слияния веток через пул-реквесты

С ветками `main` и `staging` нужно обращаться осторожно. Обычно для них устанавливают правила, запрещающие прямые отправки (push), за исключением случаев с хотфиксами, как обсуждалось в главе 25. Эти ветки связаны со средами, где простои недопустимы, — особенно `main`, так как она подключена к продакшену. Ветку `develop` тоже стоит защищать, но, возможно, с меньшими ограничениями.

Здесь самое время убедиться, что модульные тесты и сборка выполнены успешно, поскольку кому-нибудь может потребоваться подтвердить изменения в облачной среде, прежде чем менять код. Иногда нужно объединить небольшие изменения в общую ветку, чтобы не мешать чьей-то работе, и при этом добавить новые компоненты на фронтенд (например, кнопки или модальные окна) или обновить ответы от эндпоинта.

Ревью пул-реквестов

Мы проверяем код друг друга по нескольким причинам:

- чтобы убедиться, что в общие ветки попадает только качественный код;
- чтобы быть уверенными, что в продакшен не «просачиваются» явные баги;
- чтобы поделиться знаниями внутри команды;
- чтобы улучшить коммуникацию и совместную работу в команде.

Важно помнить: цель код-ревью — не критиковать друг друга и не переходить на личности. У вас может быть иной контекст, нежели у автора изменений, поэтому будьте скромнее и задавайте вопросы, если что-то кажется вам странным.

При ревью пул-реквеста стоит обратить внимание на несколько ключевых моментов. Вот чек-лист, который можно использовать.

- Потребуйте как минимум одно подтверждение от другого разработчика — это гарантирует, что новый код увидел кто-то, кроме автора.
- Убедитесь, что модульные тесты проходят, а сборка завершается успешно при отправке изменений в ветки `main` и `staging`.
- Проверьте, нет ли нарушений соглашений о коде, принятых в команде (например, именование или форматирование функций).
- Загрузите ветку локально и убедитесь, что приложение запускается.
- Запустите приложение локально и проверьте, работает ли функция так, как задумано.

- Пройдитесь по изменениям построчно и убедитесь, что вы их понимаете.
- Если остались неясности, задайте вопросы, потому что вы можете быть не единственным, у кого они остались.
- Старайтесь не навязывать свой стиль кода другому разработчику — одну и ту же задачу можно решать разными способами.
- Обратите внимание на условия в коде и вызовы сторонних сервисов.
- Перепроверьте данные и убедитесь, что они отправляются и возвращаются в ожидаемом формате.

Вы можете добавить любое количество задач (https://oreil.ly/Y8VfS) в процесс ревью пул-реквеста и даже создать шаблон (https://oreil.ly/1Ivga), который будет автоматически доступен для всех пул-реквестов. Предложенный чек-лист содержит минимальный набор пунктов, поэтому обсудите с командой, какие критерии наиболее важны именно для ваших ревью. Один из способов упростить процесс — заранее договориться о правилах работы с ветками.

Ветки для небольших фич

Как правило, каждая создаваемая вами ветка будет связана с тикетом, особенно при исправлении багов или выполнении небольших разовых задач. Перед созданием новой ветки всегда забирайте последние изменения из ветки `develop`. Лично я синхронизирую изменения ежедневно, а иногда и по нескольку раз в день, чтобы убедиться, что я работаю с актуальным кодом. Это страхует меня от использования устаревшего кода, который позже может вызвать конфликты при слиянии.

Для таких веток старайтесь делать небольшие коммиты, когда это возможно. Возьмем, к примеру, задачу по созданию формы регистрации пользователя. Она потребует новых типов данных, нового компонента, вызова API и, возможно, установки новых пакетов. Каждое из этих изменений можно, если желаете, оформить отдельным коммитом — это упростит отладку. Когда вы будете готовы мержить изменения в `develop`, будет неплохо объединить все коммиты вашей ветки в один. Подробнее об этом поговорим чуть позже.

Поскольку у вас и у команды уже есть соглашения по именованию веток и сообщениям коммитов из главы 27, цель ваших изменений становится понятнее. Эта ясность особенно полезна при обновлении пакетов в рамках задачи, так как подобные изменения могут вызвать каскадные проблемы после слияния. Вы сможете точно определить, какая функция или исправление бага содержат обновления, и отменить именно их, если они начнут блокировать других разработчиков. Даже если вы интегрируете несколько тикетов и объединяете коммиты, процесс будет выглядеть так, как показано на рис. 28.2.

Как только эти небольшие ветки получают одобрение, их следует сразу же мержить. Это поможет поддерживать поток изменений в ветке `develop` и предотвратит устаревание веток. *Устаревшая ветка* — это та, которая простояла достаточно долго,

чтобы в `develop` появились новые изменения после ее утверждения. Небольшие и быстрые слияния уменьшат количество конфликтов и помогут команде разработчиков оставаться в курсе изменений.

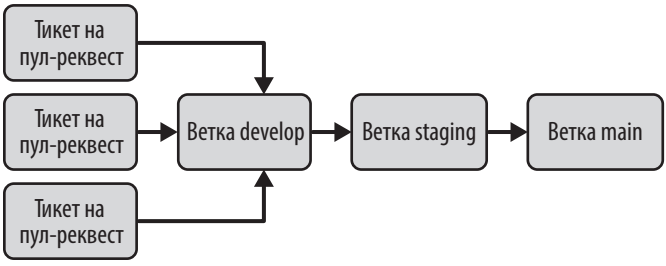

Рис. 28.2. Несколько тикетов, объединенных в ветку `develop`

Фича-ветки для крупных реализаций

Иногда в разработке находится большая задача, она добавляет значительную новую функциональность в приложение. Она может быть связана с существующим кодом или быть полностью независимой, поэтому обычно эту работу можно реализовать изолированно от других текущих изменений.

Помните, что фича-флаги, которые мы обсуждали в главе 25, также являются вариантом для такой работы. Вы можете использовать фича-флаги и функциональные ветки, если хотите сделать более детальный релиз, или выбрать один из этих подходов. При разработке крупной задачи, требующей согласованной работы нескольких частей, предпочтительнее может быть фича-ветка. Если же вы разрабатываете небольшую, но важную функцию, лучше подойдут фича-флаги.

Когда команда работает над несколькими крупными задачами, рекомендуется создавать для них отдельные ветки, в которые можно мержить небольшие изменения по мере разработки функциональности. Такой подход позволяет не блокировать ветку `develop` и дает команде возможность проверять функцию небольшими частями. Когда приходит время мержить в `develop`, отдельные части уже проходят тщательное ревью, и функционал можно оценить на более высоком уровне. Наличие отдельной фича-ветки также упрощает тестирование, так как можно сосредоточиться на одной конкретной фиче, не беспокоясь о других изменениях. Этот процесс показан на рис. 28.3.

В этом примере вы работаете над функционалом под названием *onboarding*, поэтому создали фича-ветку `feature/onboarding` на основе ветки `develop`. После создания ветки `feature/onboarding` каждая задача, связанная с `onboarding`, получила собственную ветку на ее основе. Например, ветка `task/sign-up-form` была создана на основе `feature/onboarding`, как и другие ветки задач. Эти небольшие

единицы работы проходят ревью, учитывают обратную связь, а затем мержатся в `feature/onboarding`. Таким образом, когда все задачи, связанные с `onboarding`, будут завершены, ревью пул-реквеста для функциональной ветки не составит для команды труда.

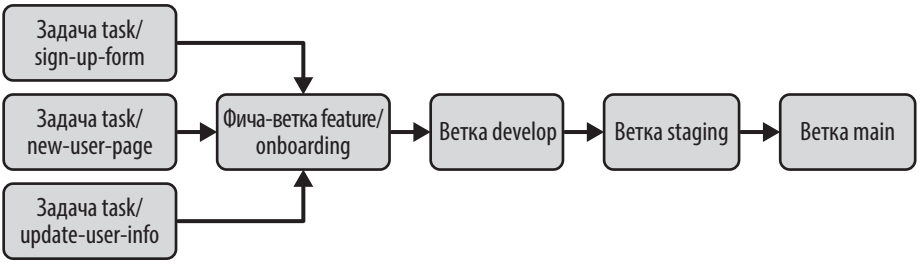

Рис. 28.3. Процесс работы с фича-веткой

Эта стратегия помогает отслеживать каждую небольшую порцию работы, необходимую для реализации конкретной задачи, позволяя сосредоточиться на одной задаче за раз и поддерживать качество кода за счет типов и тестов. Она также ускоряет время релиза, так как включает несколько этапов проверки выполнения задачи. Слияние задач напрямую в ветку `develop` может сделать использование приложения странным и неудобным, так как без использования фича-флагов вся функциональность будет еще недоступна. Использование фича-ветки дает больше времени на обдумывание пограничных случаев и вопросов, возникающих в процессе разработки, поскольку работа разбивается на мелкие части, в результате чего может выясниться, что не для всех областей сформулированы полные требования.

Объединение коммитов

Самое важное при слиянии веток — сообщать команде о завершенных слияниях. Обычно это происходит во время ревью пул-реквестов, но если изменения затрагивают общие части приложения или ключевую функциональность, стоит отдельно уведомить команду. Это позволяет совместно минимизировать конфликты между изменениями.

Существует несколько способов слияния веток, которые вы с командой можете выбрать. Первый — объединение коммитов из отдельных веток тикетов, которые мержатся в ветку `develop`. Когда вы объединяете коммиты для пул-реквеста, это означает, что все коммиты, сделанные в этой ветке, сворачиваются в один. Обычно нет необходимости видеть все мелкие коммиты, которые вошли в исправление бага или даже в функцию. Важно помнить, что сообщения мелких коммитов могут быть неформальными, как показано в примере на рис. 28.4.

Когда вы объединяете коммиты из отдельных веток в `develop`, это помогает уменьшить «шум» в истории Git, позволяя видеть изменения как целостную фичу или

исправление бага. Это полезно при отладке, так как маловероятно, что вы откатите отдельный коммит без контекста других связанных с ним изменений. Объединение мелких коммитов — это способ организовать историю по задачам и функциям или по номерам тикетов, а не по работе, которая в них вложена.

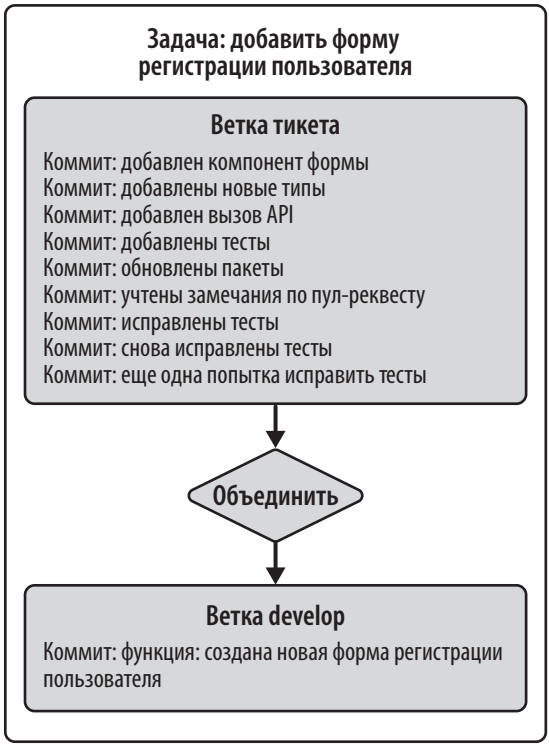

Рис. 28.4. Объединение коммитов из отдельной ветки в `develop`

Обычно коммиты, которые попадают из `develop` в стейджинг-среду (`staging`) и из `staging` в `main`, следует мержить без объединения. Поскольку вы уже объединили мелкие коммиты из отдельных веток в `develop`, в истории Git остается только самая важная информация — это помогает не потерять связь между функциями и тикетами, которые находятся в процессе тестирования или готовятся к релизу. На рис. 28.5 показано, что произойдет, если объединить все коммиты при переходе из `develop` в `staging`, и почему это затруднит понимание того, какие именно функции включены в изменения. Обычно так делать не рекомендуется.

В таком случае, если возникнет баг, вы не сможете просмотреть историю Git и определить, какая именно функция его вызвала. Если вы увидите такую запись в истории ветки `develop`, то у вас будет единственный способ узнать, какие изменения были включены: вернуться к исходному пул-реквесту. Это может привести

к потере контекста. Поэтому при слиянии изменений в общие ветки, такие как `develop`, не объединяйте коммиты.

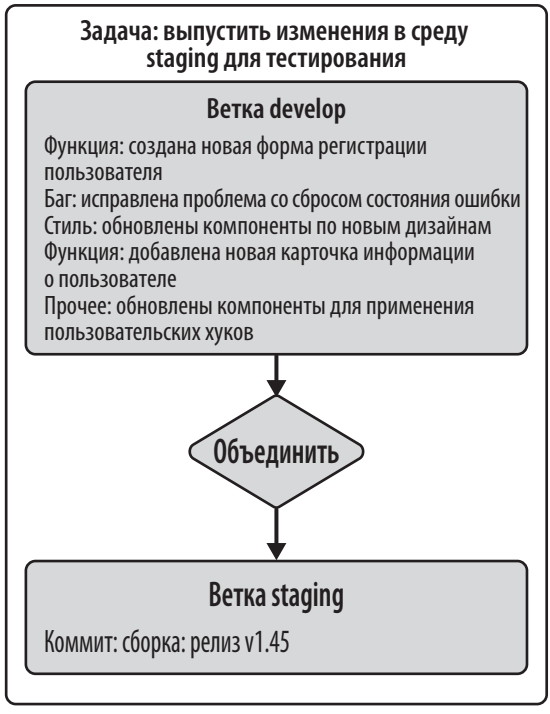

Рис. 28.5. Объединение коммитов при переходе из ветки `develop` в ветку `staging`

Крайне важно поддерживать актуальность фича-веток относительно всех изменений, которые мержатся в `develop`. Поскольку фича-ветки означают, что работа над функцией еще не завершена, они легко могут рассинхронизироваться, особенно если изменения часто попадают в `develop`. Как только ваша фича-ветка начинает расходиться с `develop`, возникают конфликты при слиянии, которые становится сложно разрешить. Минимум раз в неделю следует выполнять ребазирование (rebase) вашей фича-ветки с изменениями из `develop`. В следующем подразделе мы рассмотрим, как это правильно делать.

В качестве упражнения проанализируйте текущую историю Git в вашей организации. Пример того, как это может выглядеть в GitHub, показан на рис. 28.6.

Для объединения коммитов можно использовать команду `git rebase`, и было бы полезно разобраться, как это работает (https://oreil.ly/iMS8e). Однако на практике чаще всего для этого используются кнопки в интерфейсе пул-реквестов. Пример таких кнопок показан на рис. 28.7.

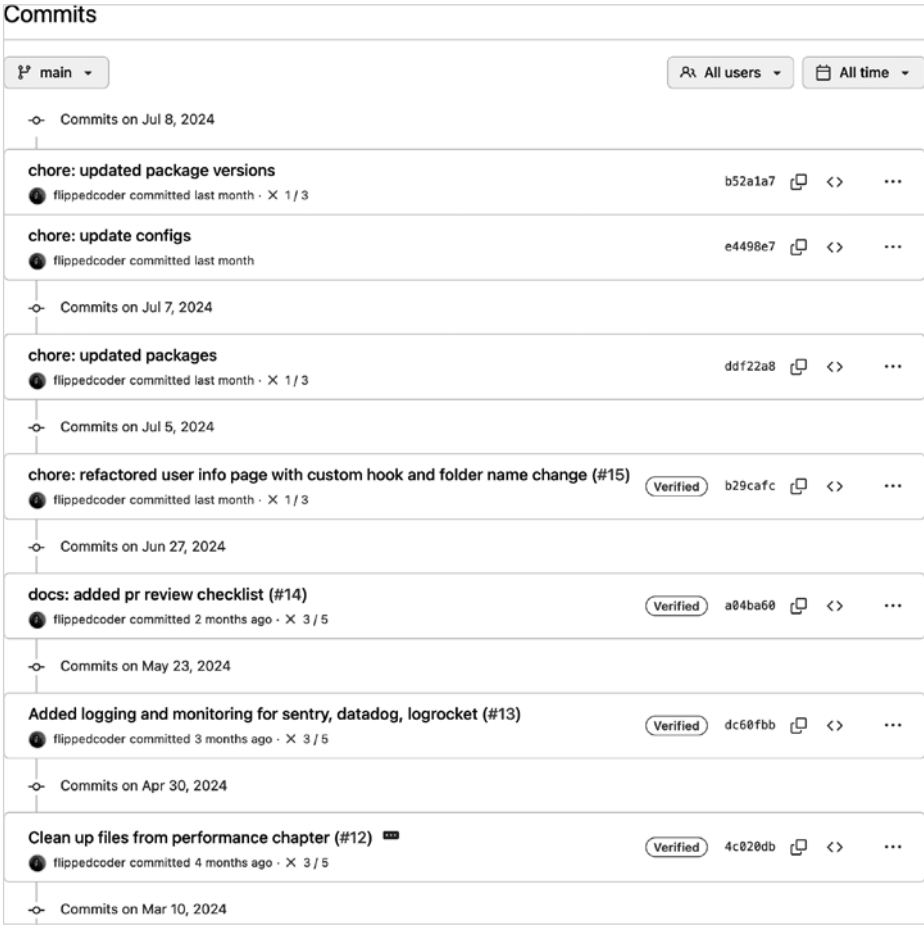

Рис. 28.6. История Git-проекта в GitHub

Выбрав опцию Squash and merge, вы можете объединить все коммиты из пулреквеста в один с сообщением, кратко описывающим изменения. Такое объединение упрощает историю коммитов и облегчает отладку. Оно также демонстрирует, почему не стоит объединять коммиты в общих ветках, — это лишает возможности видеть отдельные изменения. Если исходная история коммитов все же понадобится, ее можно восстановить, но это может быть сложно.

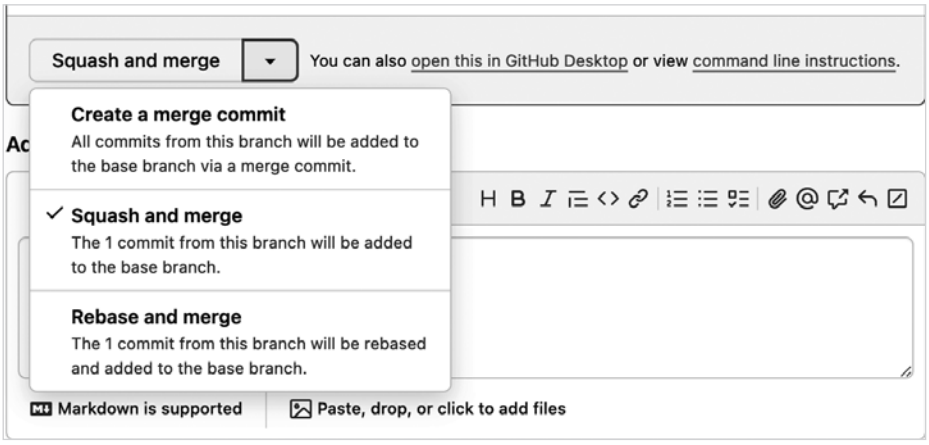

Рис. 28.7. Кнопка Squash в GitHub

Ребазирование и слияние веток

Теперь, когда вы знаете, как объединять коммиты в ветках для подготовки к слиянию в `develop`, давайте разберемся, как поддерживать фича-ветки в актуальном состоянии относительно основной ветки (например, `develop`). Это значительно упростит процесс слияния, когда изменения будут готовы к тестированию. Если долго не обновлять фича-ветку, новые изменения в `develop` могут привести к конфликтам слияния, которые будет сложно разрешить. Рекомендую в качестве общего правила: я ежедневно забираю последние изменения из `develop` и выполняю ребазирование своих локальных фича-веток. Так будет проще решать любые возникающие конфликты, и таких конфликтов будет меньше.

Вопрос актуальности фича-веток ставит перед вами и командой выбор: договориться о ребазировании фича-веток или предпочесть обычное слияние изменений из `develop`. Давайте рассмотрим различия между этими подходами и их влияние на историю Git.

Когда вы выполняете ребазирование своей ветки, это означает, что вы перемещаете начало истории фича-ветки в конец исходной ветки (например, `develop`). В результате история Git для вашей фича-ветки будет начинаться с последних изменений из `develop`, как показано на рис. 28.8 и 28.9.

Если вы и команда выбираете ребазирование, учитывайте его потенциальные сложности:

- откат изменений может стать затруднительным;
- при разрешении конфликтов слияния могут создаваться похожие, но не идентичные коммиты;
- промежуточные коммиты могут повредиться при некорректном выполнении.

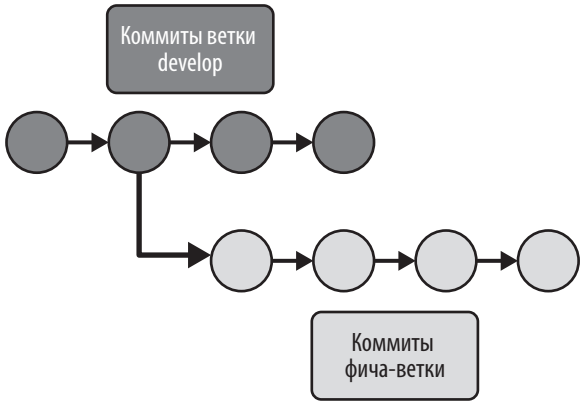

Рис. 28.8. Фича-ветка, отставшая от актуальных изменений в `develop`

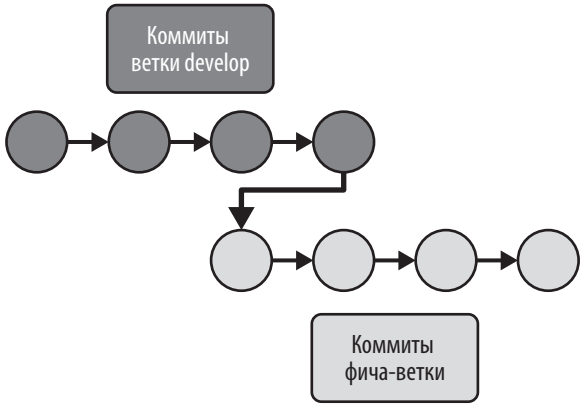

Рис. 28.9. Фича-ветка после ребазирования с текущими изменениями из `develop`

Никогда не ребазируйте общие ветки (например, `develop` или `main`) на фича-ветку — это чревато серьезными проблемами, включая удаление существующих коммитов. Ребазирование идеально подходит для индивидуальных веток, над которыми работаете только вы. Чтобы избежать осложнений, ребазируйте исключительно фича-ветки относительно общих веток.

Когда вы мержите ветку `develop` в свою фича-ветку, это означает, что вы создаете коммит с изменениями из `develop` в этой фича-ветке. В отличие от ребазирования, которое перемещает начало истории фича-ветки к последнему коммиту `develop`, слияние добавляет полную историю изменений из `develop` в начало (head) вашей фича-ветки. Таким образом, каждый раз при мерже коммита из `develop` он добавляется как последнее изменение в вашей фича-ветке, как показано на рис. 28.10. Это может быть удобно, поскольку ребазирование — более сложный процесс, тогда как

слияние выполняется в один шаг. Однако это создает проблемы, когда требуется откатить или отменить изменения.

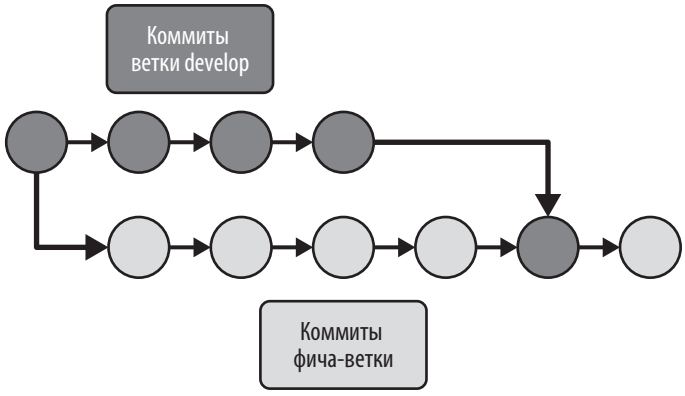

Рис. 28.10. Слияние ветки

Также важно учитывать, что делать с ветками после их слияния. Хорошей технической практикой считается удаление веток после слияния — это помогает вам и команде эффективнее управлять Git. Удаление слитых веток:

- уменьшает «шум» в репозитории;
- безопасно, так как все изменения уже находятся в общей ветке;
- зафиксировано в пул-реквесте, который был смержен.

Слияние обычно является более простым вариантом, поскольку оно просто добавляет изменения в конец фича-ветки. Вам не нужно решать, какие части отдельных коммитов сохранять в истории, как при ребазировании. Однако я рекомендую освоить и ребазирование — оно может упростить разрешение конфликтов слияния и дать более чистый результат. Оба подхода имеют свои плюсы и минусы, и вы можете комбинировать их для управления ветками. Сравнение методов представлено в табл. 28.1. Если вам нужно больше деталей о различиях между ребазированием и слиянием, ознакомьтесь с руководством Atlassian (https://oreil.ly/ddbHP), уже упоминавшейся книгой *Version Control with Git* и статьей о современных подходах к управлению ветвлением (https://oreil.ly/kjJL3).

Таблица 28.1. Сравнение ребазирования и слияния

Ребазирование	Слияние
Более линейная и чистая история	История загромождена merge-коммитами
Разрешение конфликтов может быть многоэтапным	Разрешение конфликтов выполняется за один шаг
Не рекомендуется для общих веток	Подходит для общих веток
Перезаписывает хеши коммитов	Сохраняет хеши коммитов

Совместно с командой стоит определить правила, когда использовать слияние, а когда — ребазирование. В большинстве компаний, где я работала, применялся такой подход:

- при разработке в индивидуальной или фича-ветке следует *ребазировать* ее с `develop`, чтобы поддерживать актуальность кода;
- после ревью индивидуальной или фича-ветки ее нужно *смержить* в `develop`;
- изменения из `develop` *мержатся* в тестовую среду (`staging`);
- из `staging` изменения *мержатся* в `main`.

Такой подход обеспечивает единообразие процесса работы с Git и помогает избежать ситуаций, когда приходится отменять изменения или выборочно переносить коммиты, — эти операции могут быть довольно сложными. Главное, четко понимать, какие коммиты относятся к каждой функции. Если вы можете проследить историю Git и определить, когда были развернуты функции и какой код с ними связан, значит, ваша стратегия работает.

Разрешение конфликтов слияния

Какие бы стратегии вы ни использовали, рано или поздно вам придется сталкиваться с конфликтами слияния. Это один из навыков, который будет проверяться многократно, и он поможет вам глубже изучить команды Git, улучшить коммуникацию и понимание проекта. Некоторые конфликты просты — например, несколько строк в одном файле (рис. 28.11). Другие затрагивают множество файлов с серьезными изменениями. Для их разрешения потребуется творческий подход, чтобы не потерять важные правки и не сломать приложение. Далее я расскажу о нескольких способах работы с конфликтами, которые расширят ваш инструментарий.

Обсуждение с разработчиком, который вносил изменения

Как правило, разработчик, работающий с конфликтующей веткой, должен сам исправлять конфликты. В первую очередь стоит связаться с ним и попросить его разобраться с проблемой, ведь именно он лучше всех понимает, как должна работать функция. Хотя вы, скорее всего, сможете разрешить конфликты самостоятельно, вам может не хватать контекста, чтобы определить, какие изменения стоит сохранить. Поэтому важно подталкивать членов команды к тому, чтобы они решали конфликты в своих ветках до того, как обращаться за помощью. Вы можете предложить им, например, мержить изменения из общей ветки в свою или выполнить ребазирование.

Если разработчик зашел в тупик, попробуйте организовать совместный звонок с демонстрацией экрана и вместе разобраться с проблемой. Чтобы избежать масштабных конфликтов в фича-ветках, можно порекомендовать команде ежедневно

обновлять свои ветки, подтягивая изменения из `develop`. В этом случае конфликты обычно оказываются менее серьезными или хотя бы более конкретными. Если же фича-ветка сильно расходится с `develop`, а ребазирование не помогает, можно в крайнем случае создать новую ветку с актуальными изменениями и заново добавить в нее код функции.

Рис. 28.11. Конфликт слияния в инструменте разрешения Git

Это отличная возможность обучить другого разработчика, объяснив ему, как работать с конфликтами слияния и каков процесс их разрешения. Всегда старайтесь вовлекать коллег в процесс исправления их веток, чтобы эта задача не стала исключительно вашей обязанностью. Если вам приходится вручную разбирать каждый коммит в общей ветке, привлекайте разработчика, который вносил изменения. Помните: цель не в том, чтобы обвинить его, а в том, чтобы разобраться в ситуации и исправить проблему.

Использование git bisect для поиска затронутых изменениями файлов

Иногда конфликт слияния сложно отследить через историю Git. Приходится переключаться на очень ранний коммит и проверять, присутствует ли в нем проблемный код, затем переходить к следующему коммиту в истории и снова проверять наличие конфликтующих изменений. Этот процесс надо повторять до тех пор, пока не будет найдена причина конфликта.

Именно здесь пригодится `git bisect`. Этот инструмент автоматизирует процесс: он позволяет последовательно проверять коммиты, помечая их как «хорошие» или «плохие», пока не будет найден коммит, вызвавший конфликт.

Давайте разберем конкретный пример. Сначала найдите в истории Git «хороший» хеш коммита (до возникновения конфликта) и «плохой» хеш коммита (после появления конфликта). Запустите процесс командой:

```
git bisect start
```

Теперь вы в интерактивном меню `bisect`. В любой момент можно выйти из него командой `git bisect reset`. Чтобы продолжить, укажите «хороший» коммит:

```
git bisect good b29cafce7
```

Затем введите «плохой» коммит:

```
git bisect bad f0dd190b8
```

На основе введенных данных `git bisect` переключится на коммит, находящийся ровно посередине между указанными «хорошим» и «плохим» (рис. 28.12). Так работает эта команда: она продолжит проверять коммиты на середине диапазона между помеченными как `good` и `bad`, пока не найдет проблемный.

```
● milecia@Milecias-MacBook-Pro-2 dashboard-ui % git bisect start
● milecia@Milecias-MacBook-Pro-2 dashboard-ui % git bisect good b29cafce7
● milecia@Milecias-MacBook-Pro-2 dashboard-ui % git bisect bad f0dd190b8
  Bisecting: 1 revision left to test after this (roughly 1 step)
  [e4498e79aee00896b95d57323e503364c3f995d1] chore: update configs
○ milecia@Milecias-MacBook-Pro-2 dashboard-ui %
```

Рис. 28.12. Выбор коммита между «хорошим» и «плохим»

Теперь нужно проверить файлы с конфликтами слияния, чтобы убедиться, что изменения соответствуют ожиданиям. Если нет — выполните команду, и вы увидите результат, аналогичный показанному на рис. 28.13:

```
git bisect bad
```

```
● milecia@Milecias-MacBook-Pro-2 dashboard-ui % git bisect bad
  Bisecting: 0 revisions left to test after this (roughly 0 steps)
  [ddf22a83149739f22d8681209af735c174304df7] chore: updated packages
○ milecia@Milecias-MacBook-Pro-2 dashboard-ui %
```

Рис. 28.13. Отметка следующего проблемного коммита

Такой результат сообщит инструменту `bisect`, что текущий коммит содержит ошибку, и он переключится на новый — ровно посередине между изначально помеченным как `good` и последним `bad`. Процесс повторяется до нахождения следующего «хорошего» коммита. Как только он обнаружен, выполните:

```
git bisect good
```

Затем `bisect` выведет хеш проблемного коммита (рис. 28.14), и вы сможете приступить к исправлению. Этот подход экономит часы ручного перебора коммитов и точно определяет источник конфликта. Обязательно попробуйте его в следующий раз при возникновении подобной ситуации!

Рис. 28.14. Отображение проблемного коммита и измененных файлов

Ручное сравнение изменений в файлах

Иногда конфликты слияния настолько масштабны и сложны, что ни вы, ни другие разработчики не можете определить, какие изменения следует оставить и какие последствия они повлекут. В таких случаях приходится прибегать к ручному сравнению. Сначала проверьте код в ветках `main` и `develop`, чтобы убедиться, что изменения ничего не сломают. Затем проанализируйте правки, внесенные в фича-ветках, и добавьте их вручную.

Это может потребовать создания отдельной ветки и копирования кода из нескольких веток в нее. Иногда встроенные инструменты Git для разрешения конфликтов только усугубляют путаницу, особенно при работе с большими изменениями и недостаточным контекстом. Лично я начинаю рассматривать альтернативные способы разрешения конфликтов, когда вижу, что в одном файле используется более пяти строк кода.

После ручного разрешения сложных конфликтов слияния обязательно запустите приложение локально. Иногда код может выглядеть корректно, но при выполнении возникают ошибки. Здесь пригодятся ваши тесты — они помогут выявить неожиданно затронутые изменениями участки. Все инструменты и соглашения, внедренные на ранних этапах проекта, начинают окупаться именно в таких ситуациях. Также стоит повторно проверить, что объединенные функции работают как ожидалось. На этом этапе стоит привлечь разработчиков, которые занимались этими функциями, для валидации изменений.

Заключение

Умение применять знания Git для помощи команде — важный, но часто недооцененный навык. Выбранная вами стратегия ветвления влияет на все: от CI/CD-пайплайна до повседневной работы команды. Она определяет подход к отладке сложных проблем и формирует историю всего проекта. С ростом команды неизбежно будут возникать конфликты между ветками, и коллеги часто будут обращаться к вам за помощью в сложных случаях.

Постоянно учитывайте возможности основных команд Git, таких как `rebase`, `merge` и `bisect`, которые мы разбирали в этой главе. По мере работы с конфликтами вы будете открывать новые команды и находить более эффективные способы управления ветками. Фиксируйте нестандартные подходы к разрешению конфликтов в документации. Делитесь новыми находками с командой и призывайте других делать то же самое. Так вы поможете коллегам глубже разобраться в работе Git и не останетесь единственным разработчиком, уверенно чувствующим себя в любых ситуациях.

ГЛАВА 29

Управление проектами

Ваша карьера разработчика не ограничивается только технической работой. Вам предстоит изучить основы управления проектами. Это следующий шаг к тому, чтобы брать на себя больше ответственности за то, что в итоге попадает в продакшен к пользователям. На этом этапе вы начнете задавать больше вопросов о функциях и продуктовой стратегии, а также теснее взаимодействовать с продуктовой командой. Важно научиться обсуждать целесообразность новых возможностей и аргументированно возражать, если их реализация может нарушить существующую функциональность.

По мере развития проекта продуктовая команда будет чаще привлекать вас к обсуждению того, как должны работать новые функции, какие технические аспекты нужно учесть при расширении дорожной карты продукта или какие нюансы могли ускользнуть от их внимания. От вас будут ждать экспертной оценки по вопросам того, почему реализация некоторых проектных решений займет больше времени, чем кажется на первый взгляд, в чем разница между фронтенд- и бэкенд-составляющими задач или как предлагаемые изменения могут повлиять на работу приложения. Ваша роль теперь включает не только технические экспертные знания, но и умение четко, честно и доступно доносить эту информацию до коллег.

В этой главе мы рассмотрим, как:

- участвовать в обсуждениях спринтов;
- определять и распределять задачи, чтобы команда работала в стабильном темпе;
- выстраивать коммуникацию с продуктовой командой.

Хотя этот процесс будет различаться в зависимости от компании, вы можете опираться на универсальные принципы, применимые почти везде. Главное, четко понимать границы между вашей зоной ответственности и зоной продуктовой команды, чтобы эффективнее выполнять свою роль. В моей практике были случаи, когда я слишком активно вмешивалась в продуктовые решения, и наоборот, ситуации, где стоило высказаться, но я промолчала. Обсудите с менеджером ожидания от вашей роли и убедитесь, что все участники процесса их осознают. Это избавит от множества недопониманий в будущем.

Когда я выходила за рамки своей роли, это обычно касалось определения критериев завершенности фичи. Продуктовая команда проводит обширные исследования пользователей, в которых разработчики не участвуют и о результатах которых могут не знать. Поэтому такие моменты лучше оставить им, а себе позволить задавать уточняющие вопросы. Старайтесь не строить излишних предположений о том, как фича должна работать с точки зрения пользователя, вместо этого работайте в связке с продуктовой командой, чтобы четко определить требования.

Обсуждения спринта

В ходе обсуждений спринта регулярно возникают три ключевые темы: оценка сроков, требования и распределение задач. Это командная работа, требующая совместных усилий. Хотя процесс может казаться утомительным, важно задавать как можно больше вопросов на старте — это поможет вам и команде провести спринт без неожиданных сложностей.

Оценки

Один из самых частых запросов от продуктовой команды — оценка сроков: сколько времени займут разработка, тестирование и развертывание функции в продакшен, ведь им нужно согласовать сроки с отделами продаж, поддержки клиентов и другими стейкхолдерами.

Такие вопросы часто возникают во время планирования спринта, когда команда распределяет задачи. На вас будут давить, желая быстрее согласовать сроки. Но пока у вас нет полного списка требований, готового проекта и времени на исследование, стоит воздержаться от обещаний конкретных дат.

На первый взгляд может показаться, что тикет на задачу вроде добавления нового выпадающего списка или эндпоинта прост. Но когда начинаешь вникать, оказывается, что все гораздо сложнее. Не давайте излишне оптимистичных оценок — разработка редко идет идеально по плану. Оставляйте запас времени в оценках, чтобы у команды было пространство для маневра. Вы удивитесь, с какими неожиданностями можно столкнуться даже при внесении мелких правок, например при обновлении текстов или цветов. Худшее, что может случиться, — это сорванные сроки из-за недооценки объема работ.

Хочу еще раз подчеркнуть, насколько важно быть реалистом при планировании. Когда вы хорошо знаете приложение и код, решение может прийти быстро, и это нормально. Вы можете добавить детали в тикет, если только не хотите дать другому разработчику возможность самостоятельно продумать решение. Также не стоит оценивать сроки работы, пока не обсудите задачу со своей командой.

Есть старое правило: «обещай меньше, а делай больше». Даже если вы уверены, что успеете выполнить определенный объем работы, лучше заложить чуть

больше времени на случай непредвиденных обстоятельств. Если в итоге у вас останется время на дополнительные задачи — все останутся довольны, а вы избежите лишнего напряжения. В любом случае вы исполните взятые на себя обязательства.

Скорость работы команды разработки

Потребуется время, чтобы установить базовый уровень количества стори-пойнтов, с которыми команда разработчиков может справиться. Если у вас новая команда или новый участник, может потребоваться два-три спринта, чтобы точно определить этот объем. При оценке следует закладывать буфер для перевыполнения плана. Обратите внимание, какой объем дополнительной работы выполняется, помимо тикетов по функциям и багам. Не забывайте учитывать выходные и праздничные дни. Пока у вас не будет нескольких спринтов для установки базового уровня, сложно сказать, на что действительно способна команда.

Помните, что стоимость стори-пойнтов может различаться в разных организациях. В одной компании целая функция может оцениваться в 5 пойнтов, а в другой аналогичная работа получит 13 пойнтов. Пойнты могут соотноситься с определенным временем, например, 5 пойнтов могут означать 2–3 дня работы. Однако пойнты должны отражать сложность задачи (https://oreil.ly/BRp3z) и учитывать все сопутствующие мероприятия.

Учитывайте это при выборе тикетов на спринт и оставляйте себе запас для других дел. Вам все равно придется проводить ревью пул-реквестов, исправлять баги от QA-инженеров и решать другие внезапные вопросы. Если вы возьмете максимальное количество пойнтов или задач, у вас не останется времени на эти мелкие, но важные дела. Также стоит позаботиться о других разработчиках. Во время планирования спринта, когда задачи распределяются и оцениваются, уточните у каждого, не перегружен ли он.

Тот, кто берет на себя слишком много и работает сверхурочно, легко попадает в замнутый круг, и команда будет постепенно выгорать. Здесь пригодится ваш опыт: вы знаете, какой объем работы можете выполнять без сверхусилий. Иногда хочется показать, насколько вы квалифицированны или как быстро закрываете тикеты, но это может привести к серьезным последствиям, на восстановление от которых уйдут месяцы.

Выгорание — реальная проблема. Я сталкивалась с ним несколько раз за свою профессиональную деятельность, и каждый раз требовался немалый срок, чтобы прийти в себя. У меня выгорание всегда начинается с того, что я понемногу беру на себя все больше, пока вся моя жизнь не превращается в работу. Затем проблемы накапливаются, и в конце концов я ломаюсь под давлением. Восстановиться после выгорания непросто, поэтому принимайте меры, чтобы предотвратить его (https://oreil.ly/m5_Q7).

Требования к функциям

Еще один крайне важный момент — убедиться, что у вас есть все необходимое, прежде чем соглашаться на оценку по тикету. Все дизайны и требования должны быть готовы, чтобы вы могли точно определить, сколько времени займет выполнение тикета. Согласие на неполные тикеты обычно приводит к разбуханию объема работ, и он в итоге может значительно превысить первоначальную оценку. Конечно, вопросы будут возникать в процессе разработки, но они не должны касаться основной функциональности задачи по тикету.

Также стоит предусмотреть отдельную категорию тикетов на исследовательские работы, когда детали функции еще прорабатываются. На рис. 29.1 показан пример такого тикета.

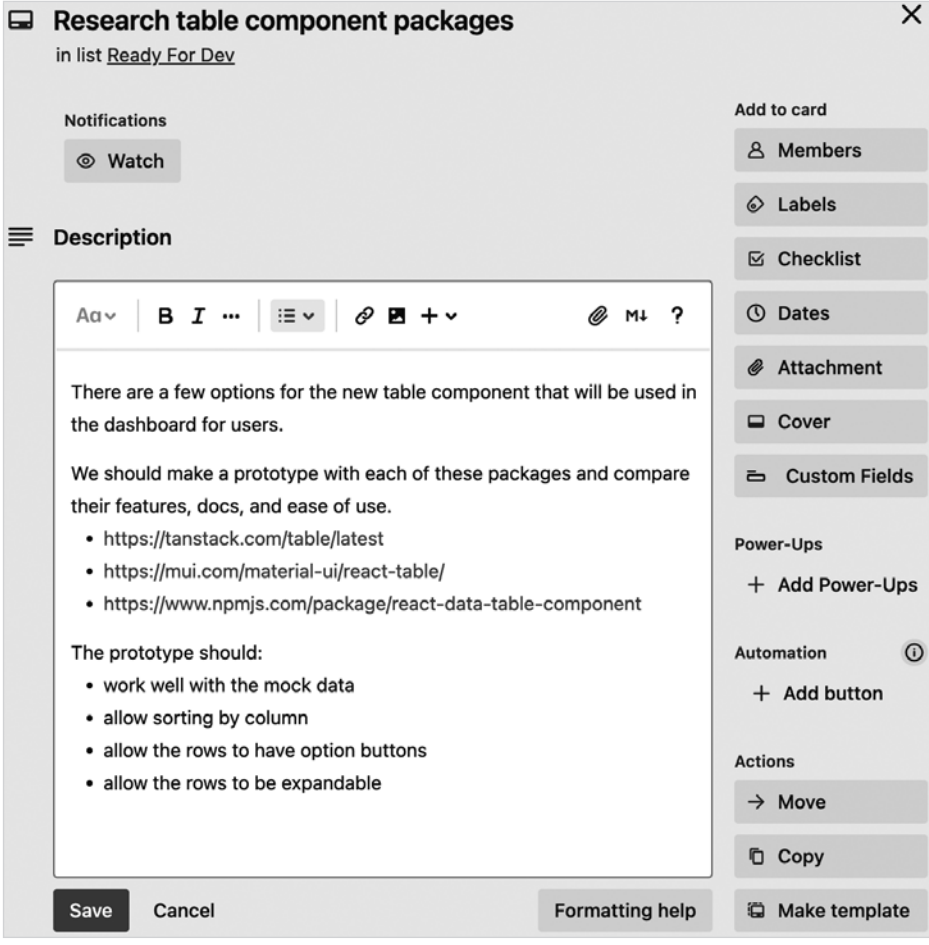

Рис. 29.1. Пример тикета на исследовательскую задачу

Если продуктовая команда не подготовила спецификации, помогите им задокументировать требования, но старайтесь не писать их вместо продакт-менеджеров. Здесь проходит тонкая грань между зонами ответственности. Спецификации должны четко описывать, как работает продукт, что он должен делать, когда это должно происходить и как это влияет на пользователя. Вы должны иметь возможность разложить эту информацию на техническую реализацию и задать продуктовой команде более детальные вопросы. Это идеальное партнерство между продуктовой командой и разработчиками. Все обсуждения в спринте должны начинаться с предоставленной вам документации. На рис. 29.2 приведен пример тикета с четко сформулированными задачами в формате Gherkin (https://oreil.ly/ae66c), который мы обсуждали в главе 21.

Жестко отстаивайте свою позицию перед продуктовой командой, если документация не готова, и не соглашайтесь на работу, требования по которой плохо сформулированы. Вы даже можете предложить использовать эпики (epics) для группировки набора тикетов, если функция достаточно крупная. Эпик (https://oreil.ly/NlmPf) — это большая функциональность, которую можно разбить на несколько пользовательских историй. Обычно он включает несколько компонентов в проекте и, возможно, требует изменений в бэкенде. Для эпика может потребоваться создать несколько тикетов, чтобы разбить работу на части, с которыми сможет справиться один разработчик или над которыми смогут параллельно работать несколько разработчиков. Так вы, помимо прочего, сможете дать более точную оценку всей функции, поскольку будете понимать ее состав.

Для средних по размеру функций может потребоваться работа над несколькими компонентами или небольшие изменения в бэкенде. Эти тикеты тоже можно сгруппировать в эпик для большей ясности, но это решение вы должны принимать совместно с продуктовой командой. Мелкие функции обычно включают несколько задач и, как правило, могут быть выполнены одним разработчиком, поэтому для них эпик не нужен. Совместно с командой разработчиков определите, как лучше разбить функцию, чтобы оценить ее масштаб.

Обзор тикетов командой разработчиков

Хорошей практикой является совместный разбор всех тикетов или спецификаций функций командой разработчиков, чтобы каждый понимал, что происходит. Это позволит вести более эффективные обсуждения с продуктовой командой. Хотя такой разбор — это еще одно дополнительное совещание, но в долгосрочной перспективе оно сэкономит общее время спринта. Организуя такие встречи, вы берете на себя роль лидера и показываете, что умеете расставлять приоритеты не только по коду.

Когда команда обсуждает тикеты с плохо сформулированными задачами, неизбежно возникают дополнительные вопросы. Разные участники команды благодаря своему опыту и взглядам помогают сделать тикет и работу более полноценными. В качестве примера того, как может проходить такое обсуждение, на рис. 29.3 показан тикет с плохо сформулированными задачами.

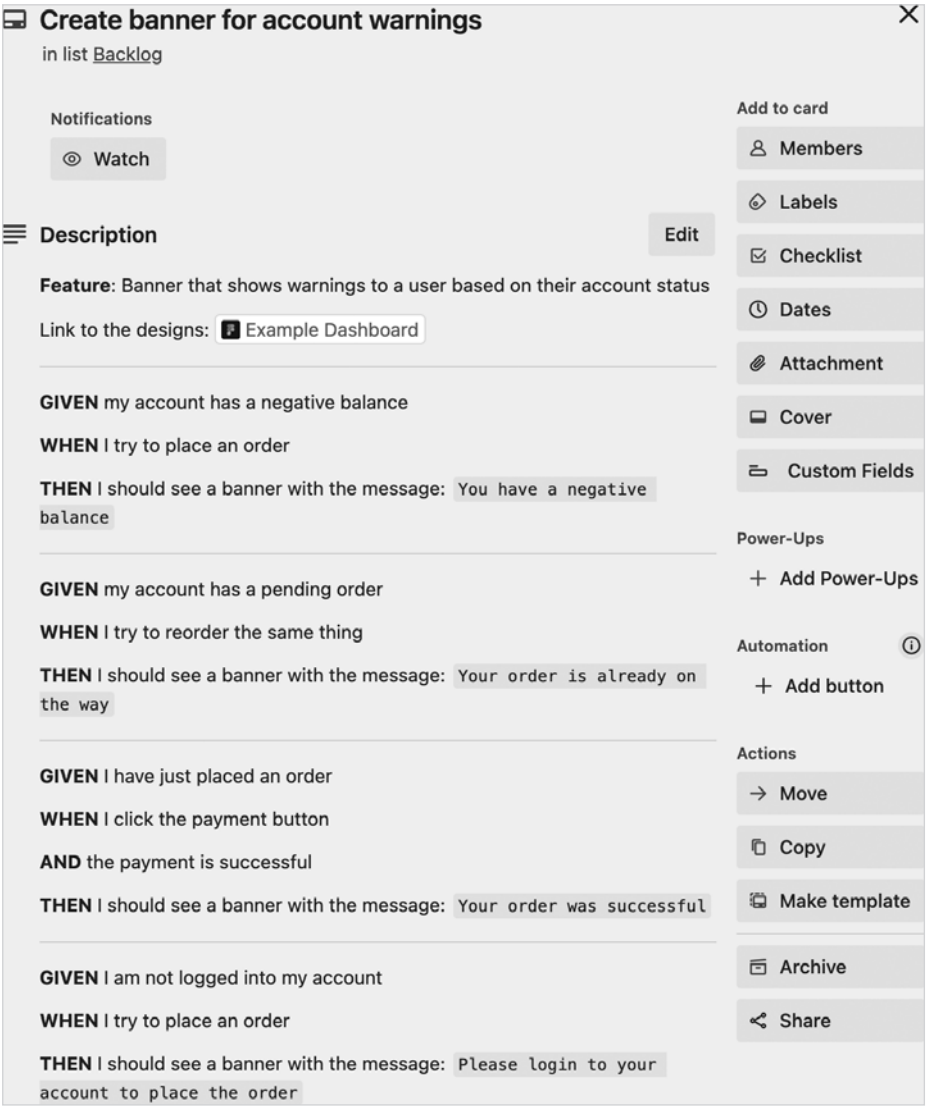

Рис. 29.2. Пример тикета с четко сформулированными задачами

Перед встречей стоит выделить время на анализ тикетов предстоящего спринта, чтобы отметить те, которые требуют наиболее детального разбора. Запланируйте как минимум час на изучение всех задач и составление списка вопросов. Первые 15–20 минут встречи отведите на то, чтобы команда могла ознакомиться с тикетами: скорее всего, у участников не было возможности сделать это заранее. Напомните команде о встрече за день до нее, чтобы люди могли выделить время на подготовку. Иногда бывает сложно вовлечь участников в обсуждение. В таком случае

задайте вопрос первым, а затем предложите каждому задать по одному вопросу. Также полезно составить чек-лист обязательных аспектов, которые должны быть определены в каждом тикете. Вот пример ключевых пунктов, которые я обычно проверяю.

- Где находятся макеты интерфейса?
- Какие данные необходимо отображать?
- Как система должна обрабатывать разные статусы?
- Как следует обрабатывать отсутствующие значения?
- Существуют ли ограничения на вводимые пользователем параметры?

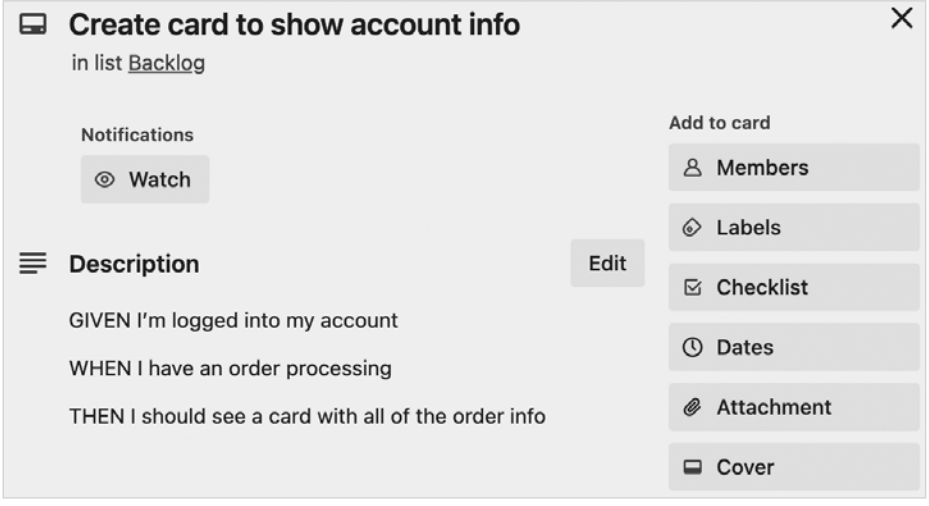

Рис. 29.3. Пример тикета с плохо сформулированными задачами до обсуждения с командой

В результате у вас накопится ряд вопросов к продуктовой команде. Отправьте эти вопросы заранее, до встречи по планированию спринта, и обязательно проконтролируйте их получение. К моменту начала планирования спринта тикет будет детализирован — и будет выглядеть примерно как на рис. 29.4.

По мере того как вы будете осваивать этот процесс, можно начать делегировать проведение встреч разработчиков другим членам команды, чтобы они тоже наработали этот опыт. Обсуждения в рамках спринта часто приводят к более глубокому пониманию функции, чем изначально предусматривала продуктовая команда, и это прекрасно. Это помогает всем участникам реалистичнее оценить:

- объем работ по реализации функции;
- оптимальный подход к разработке;
- преимущества выбранного подхода перед альтернативами.

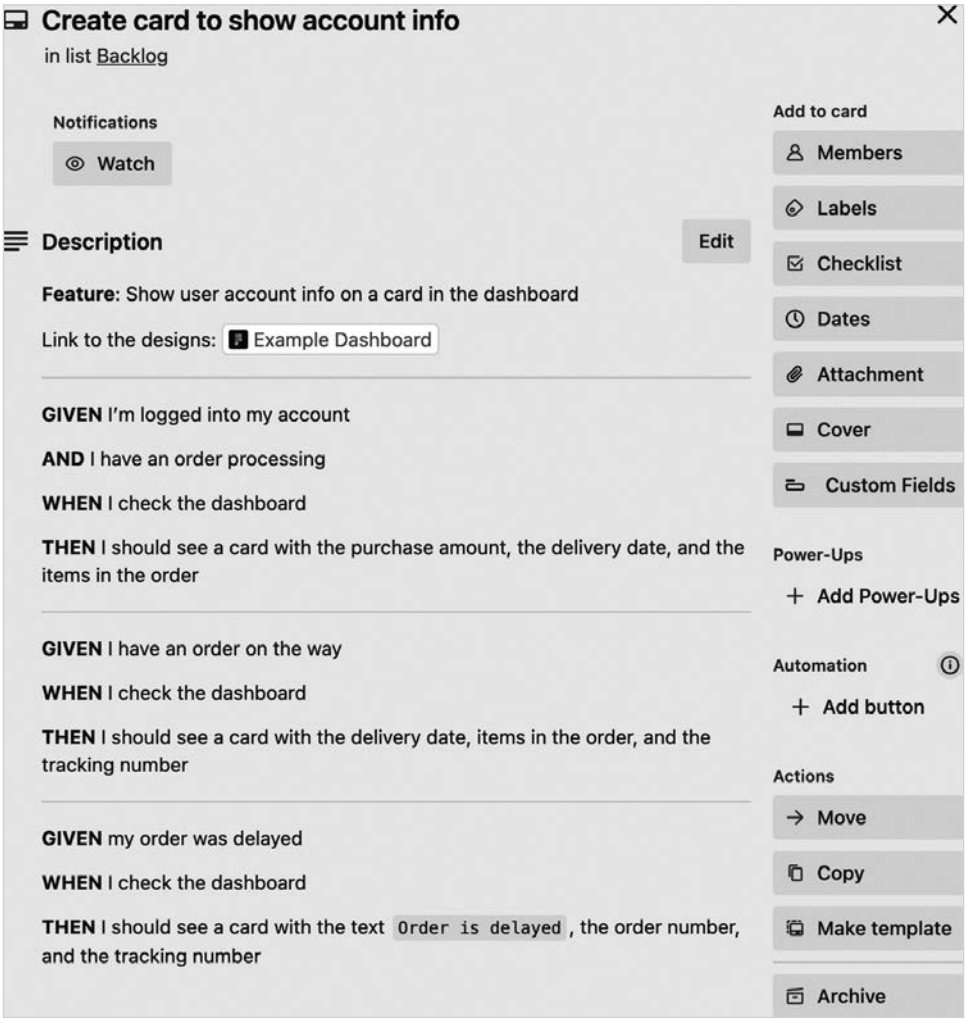

Рис. 29.4. Пример уточненного тикета после встречи разработчиков

Дорожные карты

При обсуждении предстоящего спринта всегда стоит обращаться к дорожной карте продукта. Это позволяет всем быть в курсе грядущих изменений и предполагаемых сроков релизов. Так вы избежите неожиданностей, которые для продуктовой команды уже давно не неожиданности. Регулярный просмотр дорожной карты помогает сосредоточиться на приоритетах и предусмотреть, какие задачи будут в центре внимания в следующих спринтах. Если в текущем спринте можно подготовить код для будущих функций, обсудите это с командой, чтобы получить обратную связь заранее. Пример дорожной карты показан на рис. 29.5.

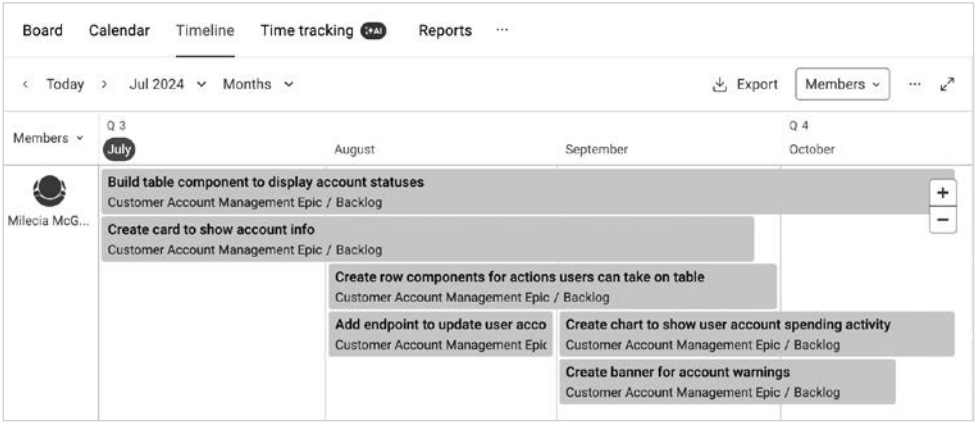

Рис. 29.5. Пример дорожной карты

Не забывайте: дорожные карты помогают нетехническим коллегам (вплоть до топ-менеджмента — вице-президентов и руководителей департаментов) понимать, над чем работает команда разработки. Это позволяет руководству быть в курсе сроков и грамотно выстраивать PR-стратегию вокруг работы команды. Легко увлечься кодом и забыть о важности дорожных карт, но именно они помогают бизнесу доносить согласованные сообщения до пользователей и стейкхолдеров.

Команде разработчиков также стоит вести свою инженерную дорожную карту, даже если это просто набор тикетов по обновлениям и рефакторингу кода. Крайне важно поднимать технические вопросы на таких обсуждениях. Когда станет известно о предстоящем обновлении SDK стороннего сервиса, мажорном обновлении пакета, необходимости рефакторинга для будущей разработки или необходимости дополнительного времени на документацию и инструменты для разработчиков, продуктовая команда должна об этом знать. Это позволит выделить время на такие задачи без давления со стороны стейкхолдеров по срокам реализации функций.

Интересный эффект совместного анализа продуктовой и инженерной дорожных карт — возможность обнаружить точки для добавления полезных функций. Визуализация всех запланированных на год возможностей в одном месте часто рождает креативные идеи. Например, продуктовая команда хочет добавить email-оповещения о действиях пользователей, а в инженерной дорожной карте уже запланировано улучшение обработки событий. Это может привести к синергетическому решению, объединяющему обе задачи. Без такого обсуждения эти функции могли бы быть реализованы отдельно и в разное время.

Определение задач и управление ими

Существует множество подходов к тому, как разработчик определяет свои задачи и управляет ими. Большинство из них основаны на личных предпочтениях, опыте

и особенностях работы организации. Стоит учитывать, как ваши задачи вписываются в дорожные карты, где возможны пересечения с работой других разработчиков, приоритеты функций и ваши собственные интересы. Эти факторы помогут сформировать индивидуальный рабочий процесс, который позволит стабильно завершать все задачи спринта и помогать команде без лишнего стресса.

Осведомленность о работе команды

Работая над функциями, старайтесь оставаться в рамках дорожной карты и иметь общее представление обо всех текущих разработках. Вам периодически будут поручать помогать коллегам с их задачами. Знать, над чем работает команда, — отличный способ приносить дополнительную пользу. Это не означает, что вы должны отслеживать все подряд и контролировать сроки исполнения, — это обязанность техлида или инженерного менеджера. Осведомленность в первую очередь сохраняет ваше собственное душевное спокойствие и помогает принимать обоснованные технические решения во время обсуждений.

Различия между техлидами и инженерными менеджерами

Техлид (tech lead, https://oreil.ly/MbEJi) — это разработчик, который наряду с основной работой берет на себя дополнительные обязанности в команде, соответствующие его экспертным знаниям. Например, может быть бэкенд-техлид, который решает наиболее сложные задачи в этой части стека и становится основным контактным лицом по соответствующим вопросам. Техлиды остаются индивидуальными исполнителями (individual contributor) и не управляют подчиненными напрямую.

Инженерный менеджер (engineering manager, https://oreil.ly/zosvG) обычно отвечает за стратегическое планирование в подразделении, бюджетирование и наем сотрудников. У него есть прямые подчиненные, и он редко погружается в написание кода. Хотя иногда он может подключаться к разработке, его основная роль — обеспечивать команду ресурсами для выполнения задач.

Например, если кто-то из команды разрабатывает новые общие компоненты, а вы в это время тоже работаете над фронтендом, могут возникнуть пересечения в работе, даже если это касается разных функций. Или если кто-то создает новую таблицу в базе данных для бэкенда, вы, возможно, сможете добавить туда свои поля и не столкнетесь с конфликтами при попытке создать то, что уже сделал коллега. Отслеживание того, чем занимаются другие, поможет всей команде избежать дублирования работы. Хотя в течение недели у вас будут встречи разработчиков, полезно самостоятельно собирать информацию о том, что происходит в проекте.

Понимание общей цели помогает команде держать ее в поле зрения. Это создает ощущение единства в работе над тикетами и показывает, к какому результату движется команда. Хотя каждый работает над своими тикетами, он лучше осознает, как его работа связана с общей целью спринта и с тем, что делают остальные участники.

В некоторых организациях продуктовая команда формирует список тикетов в спринте, исходя из их приоритета и загрузки команды. В других инженерный отдел более активно участвует в определении того, какие задачи попадут в спринт. В любом случае тикеты распределяются между разработчиками с учетом их загрузки, а иногда и специализации, например, если они больше сосредоточены на фронтенде или бэкенде. Вам может достаться крупная функция, разработка которой растянется на несколько месяцев, или множество мелких задач, охватывающих разные части системы.

Детализация и уточнение тикетов

После того как вы провели исследование, разбили задачу на более мелкие тикеты и обсудили результаты с заинтересованными сторонами, можно добавить детали в созданные тикеты. Постарайтесь включить максимум информации, чтобы любой член команды мог взять в работу ваш тикет без дополнительных уточнений. Добавьте ссылки на спецификации функции, UI-макеты или любую другую информацию, которая поможет разработчику приступить к выполнению. Также укажите первоначальные критерии приемки — это даст отправную точку для обсуждений во время планирования спринта.

Если вам интересна конкретная функция или задача — скажите об этом! Не стоит пассивно ждать назначения задач. Конечно, не всегда получится работать только над тем, что вам нравится, но в долгосрочной перспективе это сделает вас более сильным разработчиком.

Иногда вам могут поручить исследование по теме, с которой вы раньше не сталкивались. В процессе выполнения такой задачи старайтесь разбивать ее на более мелкие тикеты по мере обнаружения необходимых доработок. Не обязательно сразу прописывать все детали, но даже черновые тикеты помогут продуктовой команде лучше оценить сроки и сформировать реалистичные ожидания.

На рис. 29.6 показан пример крупной функции (эпика) с разбивкой на небольшие тикеты.

Обычно имеет смысл разделять тикеты на задачи по фронтенду и бэкенду. Бэкенд можно детализировать так: указать необходимые эндпоинты, ресурсы для создания и события, которые должны запускать действия в других системах. Фронтенд, в свою очередь, можно разбить на компоненты для добавления или обновления, API-запросы, а также хуки и вспомогательные функции. Чем более независимыми будут эти задачи, тем лучше.

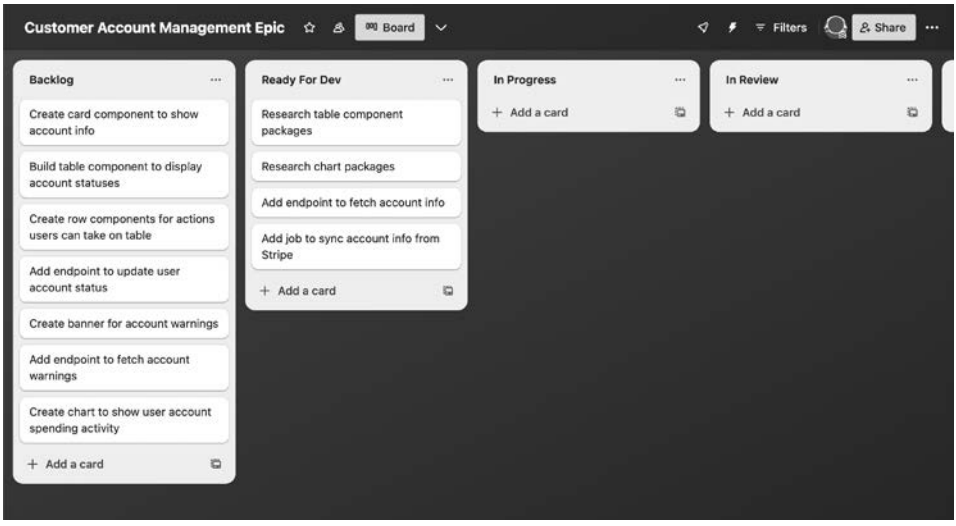

Рис. 29.6. Эпик с небольшими тикетами в Trello

Помимо разделения на фронтенд и бэкенд, полезно создавать отдельные тикеты для моков данных и тестов. Это помогает фронтенд-разработчикам понимать, что ожидать от бэкенда, а бэкенд-разработчикам — яснее видеть требования фронтенда. Так несколько разработчиков смогут параллельно работать над одной функцией без накладок. Это особенно полезно, когда кто-то завершил все свои задачи в спринте и ищет небольшие работы, чтобы помочь команде двигаться дальше. Конечно, за распределение задач отвечает техлид или инженерный менеджер, но вы можете значительно упростить их работу, если будете писать качественные и подробные тикеты.

Учет дополнительных задач

При планировании своей нагрузки важно реалистично оценивать возможности.

- Будете ли вы отсутствовать несколько дней в течение спринта?
- Потребуется ли взаимодействие с QA-инженерами для исправления багов перед релизом?
- Ожидается ли большой объем разработки, который приведет к наплыву пул-реквестов на ревью?
- Появятся ли новые члены команды или будет интеграция со сторонним сервисом?
- Планируется ли обновление версий пакетов?

Все эти факторы сокращают доступные ресурсы для выполнения новых задач. Важно заранее учитывать эти обстоятельства, чтобы не взять на себя слишком много.

Это те самые незаметные задачи, которые часто упускают из виду во время планирования спринта. Важно помнить о них и обсуждать с продуктовой командой, чтобы они понимали вашу реальную загрузку.

Такой подход подает пример для всей команды разработчиков — коллеги тоже начнут задумываться о своей дополнительной нагрузке. В результате планирование станет точнее, а продуктовая команда сможет ставить более реалистичные сроки перед стейкхолдерами.

При разборе тикетов в спринте всегда держите в голове их приоритет. Даже если какие-то тикеты кажутся вам более интересными, сосредоточьтесь на самых важных. Начинайте с высокоприоритетных тикетов — тогда в конце спринта вам не придется торопиться. К тому же при работе над ними часто всплывают непредвиденные сложности. Чем раньше вы за них возьметесь, тем лучше.

Работа в комфортном темпе

Необходимо научиться чувствовать границу между тем, чтобы решать задачи во что бы то ни стало, и тем, чтобы брать на себя слишком много. Худшее, что может случиться, — это оказаться наедине с таким количеством задач в спринте, с которым вы не сможете справиться. Если продуктовая команда просит вас сосредоточиться на новом направлении работы посреди спринта, спросите их, какое направление следует понизить в приоритете взамен. Иногда, конечно, бывают ситуации, когда необходимо справиться с повышенной нагрузкой, но они не должны восприниматься как должное каждый раз. Отдавайте себе отчет, на что именно вы соглашаетесь, потому что ваше согласие создаст прецедент, который будет трудно преодолеть со временем, и если вы будете соглашаться каждый раз, когда возникает такая ситуация, ожидание таких согласий может стать частью корпоративной культуры.

Вероятно, вы уже сталкивались с выгоранием несколько раз за свою карьеру. Восстановиться после него не так-то просто, поэтому лучше его избегать. Поддерживая разумную нагрузку, вы подаете пример команде также заботиться о себе. Как мы обсуждали в главе 22, посвященной отладке, обязательно делайте перерывы в работе в течение дня. Техника Pomodoro (https://oreil.ly/i71LF) — это популярный подход к поиску компромисса между усиленной работой и регулярными перерывами. Вы можете легко увлечься задачей, которую хочется завершить, но даже в этом случае не забывайте делать перерывы.

Управление переключением контекста

В течение дня вам, скорее всего, предстоит участвовать в нескольких встречах, отвечать на сообщения от коллег-разработчиков, разбирать вопросы от QA и продуктовой команды, работать над разными задачами и реагировать на оповещения или другие непредвиденные события. Такая активность требует постоянного переключения контекста, что замедляет рабочий процесс: каждый раз вам приходится

заново погружаться в новую задачу. Многократное повторение этого в течение дня приводит к умственной усталости и даже эмоциональной перегрузке.

Когда вы углубляетесь в работу над функцией и полностью сосредоточены, резкое переключение на нерелевантную встречу может быть болезненным. Поэтому важно защищать свое состояние потока (https://oreil.ly/070U1), когда вы в него погружаетесь. Это состояние возникает, когда вы полностью сфокусированы на одной задаче и ничто вас не отвлекает. Один из способов — использовать модифицированную технику Pomodoro: в течение выделенного времени не отвечать на сообщения и звонки. Если только в продакшене не случился критический сбой, большинство вопросов могут подождать 30 минут или дольше.

Еще один подход — разделить день на отдельные блоки. Если вы знаете, что во второй половине дня работаете продуктивнее, выделите несколько часов на сложные задачи именно в это время. На утро можно запланировать встречи, спонтанные звонки по парному программированию и «приемные часы», когда вы доступны для общих вопросов. Ближе к вечеру оставьте время на ревью пул-реквестов или другие встречи. Главное — выделить несколько часов под конкретный тип работы, чтобы не переключаться между задачами каждые несколько минут.

Признаюсь, мне потребовалось время, чтобы осознать, насколько важно защищать свое время. Раньше я считала, что на сообщения нужно отвечать мгновенно, а если коллега задает вопрос — бросать все и помогать немедленно. Такой режим работы неустойчив и несколько раз приводил меня к серьезному выгоранию. Когда я начала выделять в календаре отдельные блоки для программирования, встреч и других задач, это помогло работать эффективнее и избегать перегрузок. Помните: не каждое сообщение или оповещение требует срочного реагирования. Можно подтвердить получение запроса и сообщить, когда вы сможете его обработать.

Ищите возможности для совмещения контекстов. Например, если вы работаете над функцией с другим разработчиком, включите парное программирование в блоки фокусировки. Или попробуйте изменить рабочее окружение — это ускорит переключение между задачами. Например, перейдите в другую комнату, измените положение монитора на столе или смените фоновую музыку. Такие приемы помогают быстрее войти в состояние потока для разных задач.

Чтобы эффективно организовать день, крайне полезно несколько недель фиксировать текущую активность. Для этого можно использовать специальные инструменты (https://oreil.ly/jVsL-), например WakaTime (https://wakatime.com), или обычный планировщик. Анализ данных выявит шаблоны в вашем рабочем графике — это поможет оптимизировать расписание. Результаты могут удивить: вы ясно увидите, каким задачам и когда уделяете внимание. В течение дня записывайте, над чем работали и сколько времени потратили, отмечайте моменты переключения контекстов. Так вы соберете персональную статистику, которая подскажет, какие изменения внести.

Если вы еще не используете инструменты тайм-менеджмента, сейчас идеальный момент начать. Когда задачи на неделю четко распланированы, это снижает когнитивную нагрузку — вам не придется постоянно держать в голове, что нужно сделать. Мой любимый инструмент — бумажный планировщик. Записывая задачи на 2–4 недели вперед, я контролирую все: от рабочих вопросов до личных дел. К тому же физическое вычеркивание выполненных пунктов приносит особое удовлетворение. Если вам нужны расширенные возможности, попробуйте цифровые решения: Google Calendar (https://oreil.ly/hMVt7), Todoist (https://todoist.com), OneNote (https://www.onenote.com), Evernote (https://evernote.com) или Obsidian (https://obsidian.md).

Комбинация этих подходов поможет эффективнее переключаться между контекстами и успевать больше. Экспериментируйте с разными методами и не бойтесь придумывать собственные — главное, чтобы они работали именно для вас. Тестируйте каждый подход несколько недель, прежде чем оценить его эффективность. Найдя оптимальный вариант, придерживайтесь его! В периоды высокой загрузки у вас уже будет проверенная система, хотя иногда можно пробовать новые инструменты или техники.

Поддержание открытой коммуникации

Один из самых важных навыков — создавать культуру открытого общения в команде. Если вы понимаете, что задача займет больше времени, чем предполагалось, сразу сообщите об этом заинтересованным сторонам — это поможет избежать неожиданностей. Если в коде возникла проблема, требующая обсуждения с командой, поднимите этот вопрос. Открыто говоря о трудностях, вы показываете пример другим членам команды — возможно, они сталкиваются с теми же сложностями.

Во многих профессиональных и культурных средах вопросы воспринимаются в штыки, даже если ответа никто не знает. Старайтесь быть тем, кто не боится заговорить первым, даже если вопрос покажется кому-то глупым. Не могу сосчитать, сколько раз я тратила часы на поиск ответа, который, как мне казалось, я должна знать, только чтобы потом обнаружить, что столкнулась с такой большой проблемой, решение которой никому не известно в моей команде.

Лучше перестараться с коммуникацией и услышать мольбу: «Остановись!», чем молча бороться с проблемой в одиночку. Начинайте обсуждение с упоминания уже опробованных решений или своих идей — это помогает запустить продуктивный диалог. Ваши мысли могут выявить слабые места продукта или показать, что у коллег есть экспертные знания в неожиданных для вас областях. Помните: никто не знает всего и не стоит требовать от себя всезнайства. Если вы обнаружили противоречия в требованиях или технически рискованные решения, сразу сообщите об этом продуктовой или проектной команде.

Заключение

В этой главе мы рассмотрели методы управления проектами, которые помогут вам выстроить баланс между вашими задачами и работой продуктовой команды. Не будьте пассивным слушателем на встречах — обсуждаемые решения напрямую влияют на вашу работу. Если дорожная карта или дедлайны кажутся нереалистичными, скажите об этом сразу. Когда в тикетах нет четких критериев завершения, задавайте уточняющие вопросы.

Эффективное управление временем — ваша ответственность. Используйте инструменты планирования, делайте перерывы и фокусируйтесь на задачах, которые двигают разработку вперед. Если сложно концентрироваться, попробуйте методику из книги Дэвида Аллена (David Allen) *Getting Things Done*[1] (https://oreil.ly/W0inR). Обращайтесь за помощью к команде: коллеги с разным опытом могут дать ценные советы. Так вы постепенно наращиваете лидерские качества — ваши действия создают культуру, которая усиливает всю команду.

[1] На русском языке: Аллен Дэвид. Как привести дела в порядок. Искусство продуктивности без стресса.

ГЛАВА 30

Понимание специфики бизнеса

То, что по-настоящему выделит вас как разработчика, — это глубокое понимание специфики бизнеса, для которого вы создаете приложение. Любой разработчик может написать код по требованиям, но когда вы разбираетесь в специфике конкретной области, вы сможете принимать более обоснованные решения. Специфика бизнеса может быть связана с определенными отраслями: реклама и СМИ, энергетика, государственный сектор, финансовые услуги, здравоохранение, гостиничный бизнес, производство, логистика, розничная торговля, телекоммуникации и туризм. В некоторых случаях специфика охватывает несколько отраслей или вообще не привязана к конкретной индустрии.

Эксперт, разбирающийся в специфике бизнеса, — это человек, который досконально разбирается в своей отрасли, ее тонкостях и проблемах. Зачастую такие эксперты — это не те, кто формулирует требования к продукту, и не разработчики, которые его создают. Обычно это люди из бизнеса, которые увидели нишу в отрасли и вместе с другими (включая вас) разрабатывают решение. Хороший пример эксперта — человек, проработавший годы в здравоохранении и знающий проблемы, с которыми сталкиваются медработники и пациенты.

В этой главе я рассмотрю:

- знания, специфичные для предметной области;
- потребности приложений в разных направлениях бизнеса;
- решения по архитектуре и проектированию системы;
- подходы к обучению у других команд;
- документирование.

Вам не обязательно работать в отрасли годами, чтобы досконально разобраться в ее специфике. Но у вас должно быть искреннее желание узнать как можно больше о конкретной предметной области. Это означает, что вам придется выйти за рамки мира разработки и общаться с экспертами и людьми, которые работают в этой сфере. Исследование текущих тенденций в отрасли поможет вам получить необходимые знания. Не менее важно понимать историю отрасли — знать, какие решения в ней уже пробовали и какие результаты они дали. Это одна из тех интересных вещей, которые многие разработчики всех уровней часто упускают из виду, предпочитая углублять свои технические навыки.

Не стоит думать, что у вас есть все ответы, даже после получения новых знаний о предметной области. Эксперты в этой сфере обладают большим опытом и лучше понимают процессы и системы, чем вы. У них часто есть недокументированные системы, выполняющие определенные задачи. Нам важно понимать этапы, которые надо автоматизировать посредством приложений, но без помощи экспертов, которые подтвердят корректность работы, этого не добиться.

Знания, специфичные для предметной области

На данном этапе перед вами открывается множество путей для развития карьеры и профессиональных навыков. Независимо от того, хотите вы быть сугубо разработчиком или переключиться на управление, умение изучать специфику бизнеса станет вашим огромным преимуществом.

Каждый продукт, над которым вы работаете, предоставляет услугу клиентам организации. Компания создает этот продукт, потому что он дает ей конкурентное преимущество в той или иной форме. Это преимущество может быть даже нетехническим — его ценность может заключаться в чем-то другом. Например, логистическая компания могла найти новый способ организации инвентаризации, а приложение лишь помогает в этом. Когда вы погружаетесь в специфику бизнеса (если только эта область не связана с технологиями), не стоит полагаться исключительно на свои технические знания, потому что они вторичны по отношению к основным целям организации. Помните: вы всегда разрабатываете приложение для решения какой-то проблемы. Приложение — это инструмент доставки решения пользователям, но не обязательно само решение.

Марти Каган (Marty Cagan) в своей книге *Empowered* (издательство Wiley) написал об этом: «В сильных продуктовых компаниях технологии — это не расходы, а сам бизнес. Технологии дают силу продуктам и сервисам, которые мы предоставляем клиентам. Они помогают решать проблемы клиентов способами, которые раньше были невозможны. Будь то страховой полис, банковский счет или доставка посылки на следующий день — теперь в основе каждого такого продукта или услуги лежат технологии».

Существует несколько стратегий, которые помогут вам освоиться в новой предметной области. Я рассмотрю три из них.

Обучение у людей, работающих непосредственно в этой области

Первая стратегия — учиться у экспертов прямо на их рабочем месте. Всегда полезно начинать с тех, кто ежедневно работает в этой сфере, потому что они могут точно рассказать, чем занимаются и как. Эти люди знают, что на самом деле происходит за кулисами, чтобы все работало как часы. Например, если вы создаете приложение для производственного предприятия, сходите в цех и пообщайтесь с операторами станков и механиками, которые непосредственно производят изделия.

Вы можете подключиться к онлайн-сообществам или посещать семинары и конференции по своей теме, вступить в организации, объединяющие специалистов этой сферы. Профессиональные отраслевые ассоциации помогут вам наладить контакты с людьми, которые ежедневно занимаются этой работой. Это могут быть, например, местные отделения инженерных сообществ или неформальные встречи специалистов по работе с ЧПУ-станками.

Ваша задача — выяснить, где общаются эти эксперты, и прийти к ним. Можно начать внутри своей компании: поговорите с коллегами из отдела продаж и маркетинга, чтобы понять, на какую аудиторию они ориентируются. Главное, проявить инициативу, находиться там, где идут профессиональные обсуждения, и просто слушать. Вполне нормально, если поначалу вы не будете понимать предмет разговора или истинный смысл обсуждаемых изменений. Задавайте вопросы, когда они возникают, — вы быстро найдете людей, которые с радостью на них ответят.

Полезно изучить историю предметной области — это даст вам контекст для понимания текущего состояния. Можно начать с чтения статей о новостях отрасли, чтобы узнать, как она развивалась и кто является ключевым игроком. История может даже помочь понять цель работы вашей компании. Кроме того, такое погружение поможет вам закрепиться в роли эксперта, так как вы начнете разбираться в профессиональном жаргоне. Обладая этими знаниями, вы сможете применить свои технические навыки осмысленно, показывая, как они решают задачи предметной области. Владение знаниями также повысит вашу эмпатию при разработке функций для пользователей, поскольку вы будете четко понимать их потребности.

Прохождение курсов

Второй способ углубить знания предметной области — обучение на курсах. Сегодня в открытом доступе есть бесплатные курсы практически по любым темам. Несколько поисковых запросов — и вы найдете программы по логистике или гостиничному бизнесу. Если вам нужен структурированный подход, курсы станут отличным решением. Кроме того, курсы — это возможность изучить что-то за пределами вашей сферы. Некоторые разработчики видят в этом конфликт с поддержанием актуальности технических знаний, ведь время уходит не на код и архитектурные решения. Однако лично я считаю, что это расширяет кругозор и помогает фокусироваться на главном — продукте.

Подумайте о прохождении курсов очно — возможно, ваша компания согласится оплатить обучение. Такие программы дают практический опыт, позволяя глубже понять специфику предметной области. Хотя общение с экспертами и самостоятельное изучение материалов тоже полезны, некоторые области требуют более структурированного подхода. Рекомендую выбрать одну сферу и начать с бесплатных курсов, чтобы понять, действительно ли вы хотите погружаться в нее.

Работа в выбранной области

Третий способ получить знания — временно поработать в этой сфере. Например, можно найти волонтерские проекты. Когда-то я интересовалась строительством и присоединилась к Habitat for Humanity — некоммерческой организации, которая строит и ремонтирует дома для нуждающихся. Это был отличный способ помочь людям и одновременно многому научиться. Также можно искать частичную занятость, чтобы оставаться в IT, параллельно изучая другую область. Лучший способ, который я нашла, — предложить помощь местным компаниям, объяснив свои цели.

Возможно, вы уже задумывались, чем бы занялись, если бы не писали код, и хотели изучить другие варианты. Если у вас есть такая возможность, временная работа в новой сфере даст вам множество новых навыков. Такой способ подходит не всем и требует гораздо больше вовлеченности, чем предыдущие варианты, но он может быть увлекательным и позволит взглянуть на ИТ-индустрию под другим углом.

Смена сферы деятельности откроет перед вами совершенно новый мир и, возможно, даже вдохновит на предпринимательский путь. Один из способов придумать бизнес-идею — выявить пробелы в отрасли, а лучший способ это сделать — получить личный опыт. Даже если вы не готовы на время сменить работу или заняться волонтерством, общение с экспертами после базового изучения сферы поможет заметить болевые точки или недостающие элементы. Анализируя эти наблюдения, вы сможете найти точки пересечения ваших технических навыков и знаний в предметной области — это и наметит направление для вашего дальнейшего развития.

Архитектурные и системные проектные решения

После того как вы глубже изучите специфику бизнеса, эти знания можно использовать для проектирования архитектуры программного обеспечения. Вы можете применить чистую архитектуру (clean architecture) (https://oreil.ly/QoDVj), CRUD-архитектуру (create, read, update, and delete) или многослойную архитектуру — в большинстве случаев эти подходы работают. Однако если вы работаете в более сложной предметной области, возможно, стоит выбрать архитектуру, которая лучше соответствует ее потребностям. Между предметно-ориентированными и другими архитектурами есть некоторое пересечение, поэтому важно учитывать сложность, которую вы добавляете в шаблон проектирования приложения.

Предметно-ориентированное проектирование

Предметно-ориентированное проектирование (DDD, Domain-Driven Design) — это архитектурный подход, при котором продукт проектируется в соответствии со спецификой предметной области. Его реализация требует больше усилий на

начальном этапе, поскольку вам и команде необходимо совместно с экспертами предметной области разработать единый язык для последовательного описания функциональности. Такая разработка включает создание доменной модели, отражающей правила, процессы и сущности предметной области. Заложив эти основы вначале, вы сможете построить код, который точно соответствует потребностям бизнеса.

В этой архитектуре необходимо определить несколько ключевых элементов: ограниченные контексты (bounded contexts), сущности (entities), объекты-значения (value objects), агрегаты (aggregates) и события (events). Такой подход обычно подразумевает использование микросервисов, которые работают в рамках *ограниченных контекстов* — частей кода, разделенных по бизнес-функциям.

Сущности в этой архитектуре представляют собой объекты с уникальной идентичностью, которая не определяется их атрибутами. Обычно это объекты, атрибуты которых могут меняться со временем, и эти атрибуты необходимо отслеживать. Например, заказ (order) является сущностью, потому что имеет постоянный идентификатор (например, order ID), который не меняется в зависимости от общей суммы, клиента или товаров. Однако у заказа, скорее всего, будет атрибут статуса, который нужно отслеживать и обновлять со временем. Такие *объекты* также называют *референсными*, потому что, хотя их атрибуты могут изменяться, их бизнес-смысл остается неизменным. На рис. 30.1 показано, как может выглядеть сущность заказа.

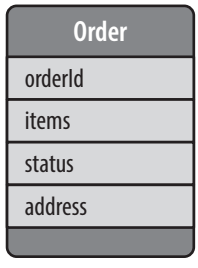

Рис. 30.1. Сущность Order

Атрибуты сущности заказа могут содержать любые значения или типы данных, но для бизнеса это всегда будет оставаться заказом. У сущностей всегда есть уникальный идентификатор, они являются изменяемыми (mutable) и обычно имеют жизненный цикл. Таким образом, хотя заказ всегда будет частью данных организации, его атрибуты могут меняться, — главное, чтобы ID оставался неизменным.

В отличие от сущностей, *объекты-значения* (value objects) определяются своими атрибутами, обычно являются неизменяемыми (immutable) и не имеют уникальных идентификаторов. К ним относятся такие элементы, как адреса, цены и даты. В нашем примере объектом-значением может быть заказчик. На рис. 30.2 показано, как выглядит объект-значение адреса.

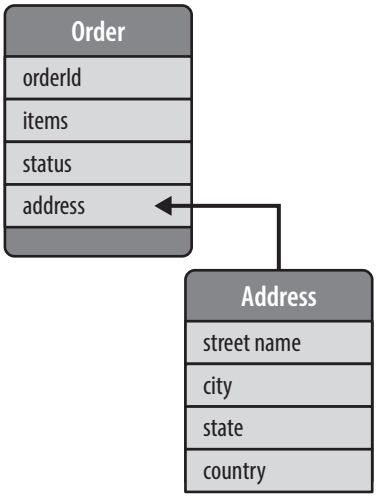

Рис. 30.2. Объект-значение Address, связанный с сущностью Order

Можно рассматривать объект-значение как тип для атрибута сущности. Если изменить один из атрибутов такого типа, это создаст совершенно новый объект. Например, если пользователь обновляет адрес доставки, мы не хотим изменять старый адрес, так как он может использоваться для выставления счетов. Вместо этого в заказ будет передан новый экземпляр адреса. Соотношение между сущностями и объектами-значениями можно сравнить с разницей между таблицей и строкой в базе данных. Сущность может представлять таблицу, тогда как объект-значение — отдельные столбцы в строке этой таблицы.

Агрегаты (aggregates) отображают взаимосвязи между сущностями и объектами-значениями, показывая, как они группируются вместе. Например, если взять заказ и добавить к нему дополнительную информацию, такую как полная сущность заказчика, заказ можно рассматривать как агрегат, поскольку происходит группировка данных.

События (events) — это триггеры в системе, которые заставляют различные части домена реагировать на изменения, происходящие в рамках ограниченных контекстов. Например, если заказчик включил автоматическое оформление заказа к определенной дате, по ее наступлении могут сработать события для списания средств и отправки товаров.

Когда у бизнеса сложная специфика, которая содержит множество переплетенных зависимостей между частями системы, имеет смысл внедрить такую архитектуру. Однако у этого подхода крутая кривая обучения: команде потребуется глубоко понимать предметную область и иметь четкое целеполагание при создании сущностей, объектов-значений, событий и ограниченных контекстов. Все эти элементы

будут усложняться по мере роста организации и расширения ее продуктовой линейки.

Необходимо гарантировать корректную обработку событий, чтобы система не оказалась в неконсистентном состоянии, где данные не обновляются должным образом или события не срабатывают в нужное время. Особое внимание этому следует уделять при интеграции со сторонними системами, поскольку данные могут рассинхронизироваться между вашей базой данных и тем, что видят пользователи в своих системах. Поддержание чистоты ограниченных контекстов также усложняется, когда вам требуется использовать данные из одного контекста для активации функциональности в другом.

Один из лучших материалов по DDD — книга Эрика Эванса (Eric Evans) *Domain-Driven Design: Tackling Complexity in the Heart of Software* (издательство Addison-Wesley). Именно эта книга в начале 2000-х заложила основы архитектуры DDD, в ней содержится исчерпывающая информация о принципах работы DDD и его внедрении в команде. Дополнительные полезные материалы по DDD можно найти в статье блога по ссылке https://oreil.ly/Nvnio. Тема DDD настолько обширна, что ей посвящено множество книг с детальными примерами. Среди них: книги Вона Вернона (Vaughn Vernon) *Domain-Driven Design* (издательство Addison-Wesley), Влад Хононов (Vladik Khononov) *What Is Domain-Driven Design?* (издательство O'Reilly) и *Learning Domain-Driven Design*[1] того же автора. Настоятельно рекомендую ознакомиться с этими материалами — в них целые главы посвящены концепциям, которые я лишь кратко упомянула здесь.

C4-модель (C4 Design)

Модель C4 (https://c4model.com) — это еще один подход к проектированию архитектуры, который можно применять при глубоком понимании специфики бизнеса вашего продукта. Вам по-прежнему потребуется совместная работа с экспертами предметной области для создания общего языка и понимания взаимосвязей между частями системы. Эта модель разбивает процесс составления диаграммы на четыре уровня:

- контекст системы (system context);
- контейнеры (containers);
- компоненты (components);
- код (code).

[1] На русском языке: Эванс Эрик. Предметно-ориентированное проектирование: структуризация сложных программных систем; Вернон Вон. Реализация методов предметно-ориентированного проектирования; Хононов Влад. Изучаем предметно-ориентированное проектирование.

Контекст системы состоит из нескольких контейнеров. Каждый контейнер содержит множество компонентов, а каждый компонент включает в себя код. Такой подход позволяет переходить от общего представления о бизнес-логике приложения к детализации отдельных частей на разных уровнях абстракции.

Контекст системы — это отправная точка, поскольку он определяет разрабатываемый продукт. Это наиболее абстрактный уровень во всей архитектуре. Здесь формулируется, как продукт вписывается в предметную область и какую проблему решает. Контейнеры в этой архитектуре определяются как:

- фронтенд-приложения;
- бэкенд-приложения;
- базы данных;
- бессерверные функции;
- системы хранения контента, такие как S3-бакеты.

К терминологии C4-модели нужно привыкнуть, поскольку понятие *«контейнер»* здесь используется в непривычном контексте. То же самое касается и термина *«компонент»* в этой архитектуре. Контейнеры похожи на то, что вы могли бы представить во фронтенд-архитектуре, но применяются ко всем частям системы. Например, ваш бэкенд можно описать через компоненты, если сгруппировать в функциональность. Так, все методы и эндпоинты, обрабатывающие заказы, в этом контексте можно назвать компонентом. *Код* (code) — это приложение, которое вы пишете для реализации компонентов. Его можно организовать в папки и файлы внутри компонентов.

Все эти элементы соединяются на диаграмме линиями, обозначающими их взаимосвязи. Это позволяет наглядно увидеть, как даже самые мелкие элементы вашего приложения вписываются в специфику предметной области. Такой подход создает архитектуру, которая работает как интерактивная карта: вы можете «приближать» и «отдалять» отдельные элементы, когда требуется уточнить детали в предметной области. Среди специализированных инструментов для построения C4-моделей можно упомянуть *Miro* (https://oreil.ly/Vwzbs), *Lucidchart* (https://oreil.ly/-5z4n), *IcePanel* (https://oreil.ly/bRzlb), *Mermaid* (https://oreil.ly/eyCyY).

Хорошим ресурсом по C4-модели является книга Саймона Брауна (Simon Brown) *The C4 Model for Visualising Software Architecture* (издательство Leanpub). На этой полезной схеме (https://oreil.ly/hGrBC) наглядно показаны различные уровни архитектуры. Главное, что нужно запомнить: C4-модель создавалась для удобного и понятного документирования работы разных частей системы.

Обучение у других команд

Чтобы глубже понять специфику бизнеса вашей организации в контексте того, что вы разрабатываете, пообщайтесь с внутренними отделами, с которыми инженеры

обычно мало взаимодействуют, например с отделами продаж, маркетинга, финансов или по работе с клиентами. Здесь важно проявить инициативу, поскольку такие возможности для общения редко возникают сами по себе.

Участвуйте в звонках отдела продаж

Разработчики обычно мало взаимодействуют с отделом продаж или самим процессом продаж. Такое происходит чаще всего в ситуациях, когда нужна техническая информация во время звонка, благодаря которой пользователи лучше поймут приложение. Такие моменты — идеальный повод поучаствовать в консультации вместе с представителем отдела продаж. Так вы сможете узнать больше о потребностях пользователей и о том, что приносит доход организации. Разговор с клиентом — отличный способ получить ценные инсайты, которые можно привнести в работу инженерной и продуктовой команд.

Во время этих звонков происходит нечто интересное: вы начинаете понимать, как создаваемое вами приложение решает задачи других организаций. Особенно это актуально, если ваш продукт предназначен для нетехнических отраслей. Возможность увидеть реальное применение продукта меняет ваш взгляд на функции и их реализацию. Кроме того, это помогает понять, как продукт презентуют люди, не вовлеченные в процесс разработки. Обсуждая технические детали, вы можете натолкнуться на идеи, которые захотите предложить команде разработки.

На этом этапе ваша задача — не техническая поддержка и не помощь с интеграцией, а участие в демонстрации продукта для отдела продаж, показ функций, о которых не знает отдел продаж, или альтернативных способов использования продукта. Однако будьте осторожны: заранее согласуйте свои действия с продакт-менеджером. Такие встречи — отличная возможность увидеть, как клиенты воспринимают ваш продукт, и понять точку зрения стейкхолдеров. Кроме того, вы лучше осознаете свою роль в компании. Вы и так знаете, что ценитесь за технические навыки, но такой опыт даст новый взгляд на разрабатываемое приложение.

Участвуйте в исследованиях пользователей

Присоединяйтесь к исследованиям поведения пользователей или наблюдайте за тестированием UX. Обычно этим занимаются UX-исследователи, продакт-менеджеры или UX-дизайнеры — в зависимости от структуры компании. Именно так продуктовая команда формулирует описание функций, над которыми вы потом работаете. Они общаются с пользователями, выявляют точки роста продукта и проводят демонстрации. Порой лучшие идеи рождаются, когда пользователь просто рассказывает, как он взаимодействует с приложением и чего ему не хватает.

Так команда дизайнеров понимает, как пользователи на самом деле реагируют на интерфейс. Например, они могут показать пользователю несколько макетов одного экрана и наблюдать, как он выполняет задачи. Такой эксперимент позволяет взять лучшее из всех вариантов и создать максимальное удобство для пользователя.

Общайтесь с отделом маркетинга

Хотя отдел продаж закрывает сделки, именно отдел маркетинга генерирует лиды. В его задачи входит запуск кампаний в соцсетях, организация стендов на конференциях, разработка и исследование портретов пользователей и заказчиков для понимания их потребностей и повышение узнаваемости компании через нетворкинг и отраслевые связи. Отдел маркетинга формирует воронку клиентов, которая ведет к продажам. Поэтому маркетологи должны хорошо разбираться в продукте, чтобы говорить о нем со знанием дела.

Пообщайтесь с кем-нибудь из отдела маркетинга — возможно, вы сможете иногда присутствовать на их встречах, чтобы лучше понять, как они говорят о продукте. Вы можете обнаружить, что некоторые части приложения им не до конца ясны и они описывают их неожиданным образом. Также вы узнаете о созданных ими портретах пользователей и получите больше информации о целевой аудитории вашего приложения. Эти знания помогут вам продумать вопросы доступности, расположения инструментов и даже соглашения по именованию.

Вы можете проверять скриншоты продукта, тексты и видео, чтобы убедиться, что они корректно отражают возможности приложения. Это еще один способ получить обратную связь от людей, которые пытаются заинтересовать других вашим продуктом. На всех встречах делайте заметки и анализируйте их позже. Возможно, выяснится, что то, что было сложно реализовать, было так же сложно понять пользователям.

Изучение звонков в службу поддержки

Изучение звонков крайне важно для разработки продукта. Понимание проблем, с которыми сталкиваются пользователи, поможет вам определить направления для улучшений. Вы также сможете помочь команде поддержки лучше разобраться в функциях приложения или создать те, что упростят им работу с клиентами. Иногда люди чувствуют неудовлетворенность, но не могут точно объяснить, что именно им нужно. Когда вы знаете, как устроено приложение «под капотом», вы быстрее увидите, где можно внести улучшения, чем пользователи смогут объяснить свою проблему.

Улучшениями могут быть небольшие изменения, например более понятная документация или соглашения по именованию, или же более масштабные, например, переработанный макет страницы, рефакторинг функции или новые инструменты. Непосредственная работа с пользователями покажет вам пограничные случаи использования, о которых вы даже не подозревали. Особенно часто это проявляется, когда пользователям нужно авторизоваться через сторонние сервисы вроде Google или Facebook. Понимание их трудностей сделает вас более эмпатичным при разработке функционала и заставит задавать больше вопросов о требованиях.

Информация, которую можно извлечь из звонков в службу поддержки, поможет вам лучше учитывать потребности внутренних стейкхолдеров. Этот аспект часто

упускают из виду в погоне за выпуском новых функций для пользователей. Однако если продукт не получает должной поддержки, эти нововведения скорее вызовут разочарование клиентов, чем решат их проблемы. Вы сами становитесь внутренним стейкхолдером, когда вам приходится реализовывать обходные решения в коде, чтобы выяснить источник проблемы пользователя.

Разберитесь в юридических аспектах

Хотя глубоких знаний в этой области не требуется, базовое понимание будет полезным. Юридический отдел следит, чтобы компания не нарушала законы, нормативные акты и стандарты соответствия. Среди них — PCI, HIPAA, GDPR и международные эмбарго. Достаточно провести пару встреч с юристами или запросить для ознакомления список юридических требований к приложению. Эти знания помогут провести технический аудит и убедиться, что ваши системы соответствуют законодательным нормам.

Поскольку вы, скорее всего, используете сторонние сервисы или инструменты с открытым исходным кодом, важно убедиться, что они соответствуют нормативным требованиям страны, в которой разрабатывается продукт. Такая проверка может оказаться сложной задачей, учитывая, что приложение создается тысячами разработчиков по всему миру. Однако имея список правил, проще понять, на что обращать внимание.

Обсуждение с юридическим отделом также поможет раскрыть неизвестные вам аспекты специфики бизнеса. Юристы, вероятно, не используют ваше приложение напрямую, но они могут объяснить его юридические последствия.

Организация документации

Хотя полная ответственность за документирование специфики бизнеса для инженерных и продуктовых команд лежит не только на вас, ваши рекомендации будут полезны. Важно помочь команде организовать документацию так, чтобы ее было легко искать. При создании документов, посвященных бизнес-логике, легко перегрузить их информацией, полученной от экспертов предметной области, других отделов и даже из личных заметок. Ниже несколько советов по поддержанию документации в порядке.

Определение терминов

Поскольку это документация по специфике предметной области, в ней могут встречаться аббревиатуры или профессиональный жаргон, который не всем знаком. Не стоит предполагать, что читатели понимают значение терминов, — эти материалы могут стать частью вводного обучения для новых членов команды, как в разработке, так и в других отделах. Четкие определения помогут команде разработчиков понять, почему архитектура системы разделена определенным образом и какие причины стоят за конкретными решениями в коде.

Вы можете кратко изложить ключевые аспекты предметной области, а затем уточнить детали вместе с продуктовой командой. Поскольку у продуктологов экспертные знания в логике предметной области обычно глубже, привлекайте их к совместному наполнению документации. Когда они добавляют новые функции, затрагивающие другую часть предметной области, попросите их четко описать ее в общих документах.

Сохранение только релевантной информации

Легко поддаться искушению включить в документацию все, что вы узнали о специфике предметной области, но лучше выделить ключевые моменты, а детали вынести в приложения, сноски или дополнительные заметки. Сосредоточьтесь на аспектах, связанных с архитектурой и функциями, чтобы новые разработчики понимали, как устроен код и почему он реализован именно так. Нет необходимости углубляться в историю предметной области или в то, как маркетинговая команда сегментирует клиентские персонажи. Важно найти баланс между предоставлением достаточного контекста для функционала и объяснением основ работы бизнес-логики. Помните: если информация была полезна вам, скорее всего, она пригодится и другим.

Обмен знаниями

После того как вы систематизировали свои знания, поделитесь ими с командой! Например, можно организовать неформальные обучающие встречи, где вы расскажете о внутренних процессах, ключевых выводах из профессиональных сообществ и о том, как логика предметной области связана с приложением. Такие встречи могут длиться 15–30 минут. Кроме того, предложите коллегам подготовить доклады на темы, которые их интересуют в рамках предметной области.

Более комфортный способ делиться знаниями — публиковать свежие новости или интересные находки из ваших исследований в чатах команды разработчиков. Например, можно рассказать о дискуссии с семинара или поделиться статьей, которая вам попалась на глаза. Можно даже ввести традицию, когда члены команды по очереди делают такие публикации: скажем, раз в два дня или хотя бы раз в неделю кто-нибудь делится в чате чем-то новым из того, что узнал.

Демонстрация влияния продукта на организацию

Когда вы обсуждаете специфику предметной области, объясняйте, как создаваемое вами и командой приложение помогает компании зарабатывать. Используйте диаграммы, наглядно демонстрирующие, как продукт закрывает пробелы в предметной области. Если включить это в документацию, инженерной команде будет проще понять, что и зачем она делает. Кроме того, это покажет остальным отделам компании, насколько важна разработка приложения.

Отделы в компаниях часто работают изолированно, поэтому бывает сложно объяснить, как ваша работа вписывается в общий контекст организации. Однако если

у вас есть подобная документация, вы можете наглядно показать ежедневный вклад своей команды. Такие демонстрации частично пересекаются с организационными схемами компании, но важно акцентировать внимание на инженерной (или конкретно вашей) команде, добавив в диаграмму детали приложения. Такие диаграммы и определения также упрощают обновление архитектуры и выбор методов, которые лучше подходят для долгосрочной поддержки продукта с учетом изменений в предметной области.

Когда вы показываете, как работа команды соотносится с глобальными целями организации, разработчики начинают лучше осознавать свою роль. В погоне за выполнением задач спринта команда может потерять связь с общей картиной. Если вы показываете, как их работа напрямую влияет на рост компании, вы значительно повышаете их мотивацию, так как они осознают ценность их навыков и вклада.

Заключение

В этой главе мы рассмотрели, как изучать специфику предметной области, и объяснили, почему это важно. Понимание предметной области, в которой вы работаете, значительно упрощает выбор инструментов и подходов к разработке. Оно также помогает глубже вовлекаться в процессы компании на системном уровне. Всегда помните: вы не просто пишете код для технически совершенного продукта — вы создаете продукт, который решает задачи пользователей и, возможно, приносит им радость.

Сочетание знаний о предметной области с навыками выстраивания отношений с другими отделами делает вас незаменимым специалистом в любой организации. Благодаря этим знаниям вы, как разработчик, будете работать более вдумчиво: начнете учитывать нормативные требования, которые могут затрагивать приложение, осознаете влияние ваших изменений на клиентов и их роль в развитии бизнеса. Обладая такими экспертными знаниями, налаженными связями и техническими навыками, вы станете универсальным senior-разработчиком, знающим все аспекты работы.

ГЛАВА 31

Работа над разными типами проектов в течение карьеры

На протяжении этой книги мы разрабатывали продукт с нуля, принимая все ключевые решения по архитектуре и реализации кода. Хотя такие проекты встречаются в практике, они скорее исключение. Чаще вы будете сталкиваться с существующей кодовой базой, устоявшимися практиками и базовой функциональностью, которую нельзя легко изменить.

Меняя организации в течение карьеры, вы будете осваивать новые навыки и паттерны, работая с уже созданными проектами. В командах с другими senior-разработчиками вы познакомитесь с альтернативными парадигмами и методами создания приложений. Когда вам снова представится возможность работать над проектом с чистого листа, подобно описанному в книге, вы подойдете к нему с бóльшим опытом. После длительной работы с legacy-системами работа над проектом с нуля создаст для вас принципиально иные вызовы.

В этой главе мы рассмотрим:

- особенности работы с совершенно новыми приложениями;
- нюансы поддержки существующих приложений;
- пути влияния этих аспектов на вашу карьеру.

Здесь вы сможете проанализировать свой опыт и понять различия между новыми (greenfield) и старыми (legacy) проектами. В зависимости от типа проекта я выделю ключевые аспекты, на которые стоит обратить внимание.

Особенности работы с совершенно новыми приложениями

При работе с новыми приложениями необходимо на ранних этапах принимать множество решений, которые определят удобство поддержки кодовой базы в будущем. Хотя впоследствии вы сможете рефакторить код и заменять компоненты и пакеты на более подходящие, наличие четко определенной основы сделает этот процесс значительно более плавным. На начальных этапах потребуется много

времени и усилий для создания диаграмм, написания документации и разработки процессов, но эти вложения окупятся по мере эволюции кодовой базы и расширения команды разработчиков.

Важно понимать, что нереально предусмотреть все возможные сценарии заранее — некоторые аспекты будут выявляться по мере разработки и получения обратной связи. Однако вы можете заложить прочный фундамент, который будет способствовать началу этих обсуждений. Вам и продуктовой команде предстоит найти баланс между предварительным планированием функциональности и ее непосредственной реализацией.

Работа с новым приложением — это ваш шанс организовать кодовую базу так, как вы привыкли, работая с другими проектами: у вас уже накоплен некоторый опыт, вы уже наблюдали типичные ошибки и видели недочеты в других организациях. Независимо от того, создаете вы приложение для стартапа, средней или крупной компании, основные принципы остаются схожими. Вам нужно учитывать инструменты, которые предпочитает организация, а затем дополнять их своим опытом. Здесь поможет чек-лист, который вы сформировали за годы работы.

Я расскажу вам о шагах, которые выполняю при создании каждого нового фулстек-приложения. Это простенький чек-лист для начала. Каждый пункт содержит множество деталей, которые мы уже разбирали в предыдущих главах этой книги.

Понимание решаемой проблемы

Прежде чем создавать новый репозиторий, необходимо четко понять проблему, которую решает разрабатываемое приложение, как мы это делали в главе 1. Для этого потребуется провести множество встреч с продуктовой командой и руководством, чтобы определить функциональные требования. Вам даже придется обсудить вопросы с коллегами, которые выше вас по должности, чтобы глубже понять специфику бизнеса, то, как они видят роль приложения в этой области, какие могут быть слабые места у приложения и каковы их планы по решению потребностей бизнеса с помощью приложения. После этого у вас будет более ясное представление о дорожной карте при обсуждении с продуктовой командой.

На этом этапе вы анализируете дорожную карту и принимаете технические решения, которые будут масштабироваться вместе с приложением. Когда вы четко поймете проблему, можно начинать выбирать инструменты и сервисы, которые лучше всего подойдут и потребуют минимального рефакторинга в будущем. Для отслеживания решений может помочь создание списка программных компонентов (SBOM, software bill of materials), где фиксируется, какие инструменты используются и какие технические задачи они решают, их влияние на безопасность и степень интеграции между собой.

Генерация простого SBOM с помощью AuditJS

Существует множество инструментов (https://oreil.ly/uzDfl) для создания SBOM различного уровня детализации, но одним из самых быстрых для использования в репозиториях является AuditJS (https://oreil.ly/6kOW3). Этот инструмент генерирует список всех зависимостей в репозитории, их текущие версии и имеющиеся уязвимости. Пример такого отчета показан на рис. 31.1.

```
[383/486] - pkg:npm/resolve@1.22.8 - No vulnerabilities found!
[384/486] - pkg:npm/restore-cursor@3.1.0 - No vulnerabilities found!
[385/486] - pkg:npm/rfdc@1.4.1 - No vulnerabilities found!
[386/486] - pkg:npm/rollup-plugin-visualizer@5.12.0 - No vulnerabilities found!
[387/486] - pkg:npm/rollup@4.20.0 - 1 vulnerability found!

  Vulnerability Title:  [CVE-2024-47068] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scriptin
g')
  ID:  CVE-2024-47068
  Description:  Rollup is a module bundler for JavaScript. Versions prior to 2.79.2, 3.29.5, and 4.22.4 are susceptible to a DOM C
lobbering vulnerability when bundling scripts with properties from `import.meta` (e.g., `import.meta.url`) in `cjs`/`umd`/`iife` f
ormat. The DOM Clobbering gadget can lead to cross-site scripting (XSS) in web pages where scriptless attacker-controlled HTML ele
ments (e.g., an `img` tag with an unsanitized `name` attribute) are present. Versions 2.79.2, 3.29.5, and 4.22.4  contain a patch
for the vulnerability.
  CVSS Score:  6.1
  CVSS Vector:  CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:C/C:L/I:L/A:N
  CVE:  CVE-2024-47068
  Reference:  https://ossindex.sonatype.org/vulnerability/CVE-2024-47068?component-type=npm&component-name=rollup&utm_source=auditj
s&utm_medium=integration&utm_content=4.0.45

[388/486] - pkg:npm/rxjs@7.8.1 - No vulnerabilities found!
[389/486] - pkg:npm/safe-buffer@5.2.1 - No vulnerabilities found!
[390/486] - pkg:npm/safer-buffer@2.1.2 - No vulnerabilities found!
[391/486] - pkg:npm/scheduler@0.23.0 - No vulnerabilities found!
[392/486] - pkg:npm/seed-random@2.2.0 - No vulnerabilities found!
[393/486] - pkg:npm/seedrandom@3.0.5 - No vulnerabilities found!
[394/486] - pkg:npm/semver@5.7.2 - No vulnerabilities found!
[395/486] - pkg:npm/semver@7.5.3 - No vulnerabilities found!
```

Рис. 31.1. Пример SBOM

Построение схемы данных

В главе 3 мы увидели, что выбор модели данных — это важный этап, поскольку он влияет на применяемые инструменты и будущее масштабирование приложения. Независимо от того, используете вы SQL или NoSQL базу данных, в схему следует включать стандартные аудит-поля: дату создания (`"created at"`), дату обновления (`"updated at"`) и идентификатор пользователя, внесшего изменения. Многие ORM-фреймворки автоматически добавляют эти поля, которые применимы к любой создаваемой таблице. Также необходимо продумать взаимосвязи между данными, чтобы выбрать наиболее производительный способ их хранения и обращения.

Для визуализации и организации схемы данных, типов данных и их взаимосвязей (как в главе 3) полезны инструменты вроде dbdiagram.io (https://dbdiagram.io) или Miro (https://miro.com). Работайте планомерно: сверяйтесь с дорожной картой при разработке начальной схемы. Учитывайте масштабируемость, так как со временем система может интегрироваться с хранилищами данных или пайплайнами обработки. Также соблюдайте единые соглашения по именованию — это сделает структуру данных понятной и точной.

Выбор архитектуры системы

Выбор архитектуры системы — это критически важное решение, поскольку команда разработчиков будет привязана к нему на долгое время. Как показано в главах 5, 6, 10 и 23, следует выбирать простую, но гибкую архитектуру для старта проекта. Доступные варианты включают различные комбинации: монолитный бэкенд, микросервисы, монолитный фронтенд и микрофронтенды. При выборе архитектуры стоит учитывать специфику бизнеса и текущую дорожную карту, чтобы прогнозировать будущее развитие приложения. Важно создать детальные диаграммы, отображающие все взаимосвязи, и регулярно их обновлять.

Хотя проектирование архитектуры — трудоемкий процесс, тщательная проработка на начальном этапе помогает выявить потенциальные проблемы заранее. Рекомендуется сначала построить полную диаграмму, а затем постепенно ее уточнять. На диаграмме следует отразить фоновые задачи, воркеры, события, подключения эндпоинтов к фронтенду и сторонние сервисы. Не обязательно сразу продумывать все детали приложения, но хорошая оценка архитектуры даст команде надежную основу для принятия дальнейших решений.

Выбор облачного провайдера

Облачный провайдер определяет многие вещи — от сервисов, которые будут вам доступны, до размера счета, который организация будет ежемесячно оплачивать. Проведите сравнительный анализ стоимости трех-четырех провайдеров, учитывая необходимые вам сервисы. Будьте реалистичны в оценке текущей и прогнозируемой пользовательской базы. Изучите дополнительные сервисы, интегрируемые с облачными платформами. Сравнительная таблица с вашими требованиями поможет сделать экономически обоснованный выбор.

Также учитывайте навыки разработчиков и их готовность осваивать новые технологии, поскольку команда должна уметь работать с выбранными инструментами. Особое внимание уделите вопросам безопасности, особенно в регулируемых отраслях. Продумайте сложность потенциальной миграции на другую платформу. По возможности протестируйте сервисы разных провайдеров, чтобы оценить удобство конфигурации и доступность поддержки.

Разработка бэкенда

На этом этапе вы приступаете к реализации спроектированной архитектуры. Как мы обсуждали в главе 2, можно начать с выбора инструментов для кодовой базы, которые упростят разработку. Полезно создать каркас проекта — стандартные папки и структуру файлов, соответствующие вашей архитектуре и схеме данных. Такой каркас даст команде четкое понимание, как начинать добавлять новые функции.

Подключите приложение к базе данных и создайте скрипт для заполнения тестовыми данными, как это делалось в главе 3. Также стоит разработать пару эндпоинтов для первичного тестирования. Начните с небольших функций, например: эндпоинт

для получения заказа из базы данных и эндпоинт для обновления профиля клиента. Главная цель — получить работающую систему, убедившись, что все ключевые подключения, права доступа и переменные окружения настроены корректно, до того, как команда разработчиков сосредоточится на написании кода.

Разработка фронтенда

Аналогично бэкенду, вам предстоит реализовать архитектуру фронтенда и определить структуру приложения. Если вы выбрали монолитную архитектуру фронтенда, начните с организации структуры папок для компонентов, вспомогательных методов и вызовов API. Для микрофронтенд-архитектуры, как обсуждалось в главе 13, создайте отдельные репозитории, которые станут компонентами общего фронтенда. На этом этапе также выбираются инструменты фронтенда, определяется способ вызова эндпоинтов и обработки стилей.

Для проверки работы приложения попробуйте создать основной контейнерный компонент и небольшой тестовый компонент. Затем выполните API-запрос из этого небольшого компонента, чтобы убедиться в корректном подключении к бэкенду. Проверьте базовую адаптивность ваших дизайнов и начните формировать общую функциональность. Добавление тестов на этом этапе заложит основу для согласованного уровня покрытия кода.

Интеграция бэкенда и фронтенда

После выполнения первоначальных проверок настройки фронтенда вы с командой начнете добавлять новые функции. Даже при наличии тщательного планирования, документации и схем все равно могут возникать неожиданные проблемы с подключением, подобные тем, что мы рассматривали в главе 23. Вы столкнетесь с такими ситуациями, как отсутствие разрешений API для фронтенда, несоответствие имен и типов данных между фронтендом и бэкендом, а также проблемы с CORS (https://oreil.ly/4Vcf4). Именно на этом этапе обычно выявляются участки кода с обеих сторон, требующие рефакторинга.

В США как-то был один громкий инцидент (https://oreil.ly/0sHJ6): выяснилось, что команды, работавшие над формой FAFSA в рамках федеральной студенческой программы, не провели упомянутые выше проверки. Конечно, команда фронтенда может работать в какой-то мере изолированно от бэкенда (и наоборот), но даже при такой разработке необходимо поддерживать постоянное взаимодействие между командами.

На этом этапе вы получаете полное представление о том, как будет развиваться приложение, и о том, какие решения команде предстоит принять в будущем. Обновляйте архитектурные схемы по мере выявления сложных участков и новых взаимосвязей между фронтендом и бэкендом. Это подходящий момент для установления более строгих правил и соглашений по проекту, а также для оптимизации проверки пул-реквестов и процесса развертывания.

Настройка CI/CD-пайплайна

В главе 27 вы создавали простой CI/CD-пайплайн. Этот процесс включает настройку сред для развертывания фронтенда и бэкенда, а также определение места хранения секретных и учетных данных. Хотя этим обычно занимается команда DevOps, от вас ожидается помощь в проверке корректности развертывания приложения. Вам нужно будет проверить, какие эндпоинты вызываются, чтобы убедиться в правильном подключении сред на фронтенде и бэкенде, а также выполнить проверки данных для подтверждения их происхождения из нужной среды.

Проверьте свои облачные сервисы, чтобы убедиться, что ресурсы находятся в правильных локациях и настроены с соответствующими уровнями доступа. Настройте автоматизацию для ваших веток, которая поможет стабильно выпускать изменения. Проверьте размер сборки и поищите возможности для оптимизации. Если вы используете контейнеры, убедитесь, что они правильно настроены для ваших сред.

Тестирование с командой QA

Совместно с командой QA протестируйте приложение и напишите тест-кейсы, покрывающие всю функциональность, требуемую продуктовой командой, а также найденные пограничные случаи. Приглашайте продуктовую команду на показы по мере разработки и развертывания функций для проведения тестирования в рамках пользовательской приемки. Также обсудите автоматизированное тестирование как задачу, которую можно разделить между QA и командой разработчиков.

Если в организации нет выделенной команды QA, вы можете проводить тестирование самостоятельно на основе спецификаций функций и дизайнов. Проверьте работу приложения в разных браузерах, при различных скоростях сети и для разных типов пользователей. Намеренно вызывайте ошибки в приложении и оценивайте, насколько хорошо оно с ними справляется. Сравнивайте данные, полученные из бэкенда, с тем, что отображается на фронтенде, чтобы выявить расхождения. Убедитесь, что синхронизация данных происходит в ожидаемое время и таблицы обновляются корректно.

Проверка приложения в продакшене

При первом развертывании приложения в продакшене вам потребуется внести корректировки. Возможно, некоторые конфигурации или ресурсы нужно будет обновить, чтобы они соответствовали нагрузке в рабочей среде. Также может оказаться, что роли пользователей настроены только в других средах и их нужно добавить в продакшен. Это подходящий момент для проверки безопасности приложения: проанализируйте сетевые ответы и убедитесь, что утечек персональных данных нет. Попробуйте получить доступ к защищенным ресурсам и проверьте, реально ли приложение их защищает.

Смотрите на вещи с точки зрения пользователя, потому что именно он будет взаимодействовать с приложением. По мере подключения большего количества

сторонних сервисов вы заметите, что некоторые данные можно протестировать только в продакшене. Создайте тестовые аккаунты и пользователей или реализуйте переключение между ролями в продакшене, чтобы избежать использования реальных пользовательских данных. Проверяйте логи, чтобы убедиться, что события записываются, а оповещения работают корректно.

Особенности работы с существующими приложениями

Существующие приложения часто называют «унаследованными» (legacy). Это кодовая база, созданная до вашего прихода в команду, и такие приложения будут попадаться в вашей работе чаще всего. Обычно компании ожидают, что вы займетесь добавлением новых функций, поддержкой кода и внесением улучшений. Здесь все становится менее очевидным, поскольку не у каждой команды есть полноценная документация или четкое понимание решаемой проблемы. Вы можете предложить новое решение проблемы, которую все уже привыкли игнорировать.

Получение доступа к необходимым сервисам

Любое существующее приложение уже подключено к соответствующим сервисам, поэтому вам потребуется доступ ко всем из них. Это часто упускают из виду, пока не осознают, что не могут выполнить нужное действие. Обычно именно так выявляется, к каким сервисам нужны учетные данные, но это не самый эффективный подход. Запросите доступ ко всем сервисам при первоначальной настройке проекта и убедитесь, что он есть во всех средах.

Вам потребуется доступ к облачной платформе, системам логирования, хранилищ, мониторинга и сторонним сервисам вроде Stripe или Twilio. Если документация по подключению к этим сервисам отсутствует, дополните документацию для онбординга соответствующими инструкциями. Свежие знания помогут вам зафиксировать полезные советы, которые упростят процесс для следующих разработчиков.

Запуск среды разработки

Это одна из первых задач, которую вам предстоит выполнить, когда вы придете в команду. Вам потребуется установить необходимые инструменты, настроить сервисы и изучить особенности, чтобы запустить приложение локально. После того как вы клонируете репозиторий и установите все пакеты, проверьте документацию приложения — там могут быть дополнительные инструкции для начала работы.

Если документации окажется недостаточно, обратитесь за помощью к другим членам команды для завершения настройки. Как я уже упоминала, обновление или создание такой документации значительно упростит адаптацию новых разработчиков, поэтому вам за такую работу будут благодарны. Вы наверняка обнаружите упущенные моменты начальной настройки: локальные переменные окружения, команды, которые нужно выполнять в определенном порядке, и т. д. Убедитесь,

что у вас локально подключены фронтенд, бэкенд и база данных, чтобы иметь возможность работать со всеми частями приложения.

Изучение приложения в продакшене

Лучший способ понять, как работает приложение при взаимодействии с пользователями, — это попробовать его в продакшене. Поскольку вы начинаете работу с минимальным пониманием того, как все должно функционировать, это идеальное время задать вопросы продуктовой команде о продукте, в который вы погружаетесь. Обязательно используйте тестовые учетные данные в продакшене — нельзя под реальными учетными данными выполнять в приложении какие-то действия.

Вы также можете обратиться к команде QA-инженеров, чтобы получить тестовые учетные записи, которые они используют для продакшена. Быстрый обзор приложения в рабочей среде поможет проверить, есть ли у вас доступ к функционалу, который потребуется для будущей отладки. При таком подходе к изучению приложения команде разработчиков придется предоставить вам всю необходимую информацию, которая была упущена в документации для онбординга. Также стоит уточнить, где и как хранятся пользовательские данные, — эта информация пригодится при отладке.

Изучение приложения в непродакшен-средах

На определенном этапе нужно убедиться, что у вас есть доступ к версии приложения в роли разработчика. Это может быть стейджинг-среда, среда разработки или другие среды, возможно, даже все сразу. Это еще одна контрольная точка для проверки: достаточно ли у вас прав доступа ко всем системам и функциям, необходимым для работы. Здесь можно обратиться к другому разработчику, чтобы он провел вас по разным средам и объяснил процесс релиза. Обязательно делайте заметки: скорее всего, потребуется выполнить несколько шагов для переключения между средами.

Попробуйте найти небольшие тикеты, которые можно быстро закрыть, — это позволит проверить, отражаются ли ваши изменения во всех средах и соблюдаете ли вы правильную последовательность шагов релиза. Так вы сможете оценить, сколько времени занимает развертывание, какой уровень покрытия кода тестами есть у приложения, а также узнать о настроенных автоматических проверках безопасности. Также стоит выяснить, есть ли в системе административные страницы для внутренних пользователей или где-то используются фича-флаги. Это отличная возможность узнать больше о внутренних решениях, которые команда использует для тестирования сценариев валидации продукта или демонстраций.

Изучение кодовой базы

Изучение поможет вам понять структуру папок и архитектуру кодовой базы. Когда вы начинаете изучать код, двигайтесь в обратном направлении — от элементов,

которые видите в пользовательском интерфейсе. Такой подход проведет вас через весь стек — от фронтенда до базы данных и других частей инфраструктуры. Повторите это для нескольких функций, чтобы систематизировать свое знакомство с кодом. Некоторые разработчики пытаются просто просматривать папки и их содержимое, но такой метод не всегда дает достаточно информации для понимания контекста.

Попробуйте реверс-инжиниринг некоторых функций, чтобы выявить используемые командой подходы и паттерны реализации. Это поможет сформулировать более точные вопросы и разобраться в процессе отладки. Также полезно сопоставлять код с технической и продуктовой документацией — это покажет, как между собой связаны разные инженерные команды и чего можно ожидать от их кодовых баз.

Вы также можете попросить кого-то из команды провести вам экскурсию по коду. Задавайте вопросы в реальном времени по мере того, как начнете понимать общую логику. Когда вы получите свои первые несколько тикетов, рассмотрите возможность парного программирования с кем-то, кто уже давно в команде. У вас может быть свое видение реализации, но иногда оказывается, что уже есть устоявшийся способ решения подобных задач.

Составление заметок о потенциальных рефакторингах

По мере знакомства с кодом вы будете замечать места, которые можно было бы реализовать иначе. Если вы видите участки кода, которые можно оптимизировать для повышения производительности, или пакеты, способные упростить сложную функциональность, — отмечайте это. Работая с разными кодовыми базами, вы можете обнаружить, что у них много общего функционала. В таком случае предлагайте создать общую кодовую базу для этого функционала — именно так появляются внутренние UI-библиотеки и SDK.

Когда вы заметите вещи, которые сделали бы иначе, спросите команду, почему они реализованы именно так. Возможно, есть веская причина использовать старую версию пакета или менее популярный шаблон программирования. Вам нужен контекст принятых решений, поэтому делайте заметки о том, что вы бы изменили, и обсуждайте это с командой. Получив контекст, вы сможете предлагать решения для оптимизации производительности, повышения безопасности и удобства разработки.

Вы можете предложить, например, шаблонную кодовую базу для новых проектов, чтобы обеспечить согласованность между ними, или новые роли/права доступа для упрощения работы с ресурсами и инструментами. Добавление типов в TypeScript-проекты — еще одна область для улучшений. Как показывает опыт, в проектах, где часто меняются разработчики, нужно более тщательно продумывать структуру папок и организацию кода. Считается нормальным, если на ранних этапах в таких проектах, когда работа над первыми функциями только начинается, вы будете выступать в роли лидера.

Документирование вопросов и ответов

Если вам сложно понять, как работает определенная часть приложения, знайте: вы не одиноки. Такие вопросы часто возникают в процессе онбординга и настройки локального окружения для работы над первой функцией. Когда у вас появляются вопросы, создавайте документ и фиксируйте в нем ответы, полученные от членов команды. Как показывает практика, иногда никто не помнит, как именно настраивалась система, и приходится искать ответы в случайных Google Docs или переменных окружения у коллег.

Пожалуй, самое хорошее, что можно сделать для существующего приложения, — добавлять документацию по непонятным частям системы. Сюда может входить техническая документация, продуктовая документация и любые другие материалы, которые помогут следующим разработчикам. Не стесняйтесь задавать вопросы на протяжении всего периода адаптации и после него. Вы гораздо быстрее поймете, как работает организация, если будете активно спрашивать, а не ждать, пока кто-то спросит, есть ли у вас вопросы.

Повышение качества кода

Такое повышение означает применение принципов DRY (Don't Repeat Yourself — «Не повторяйся») при обнаружении дублирования кода и создании общих служебных функций и компонентов. Не бойтесь обсуждать с командой используемые шаблоны кода и причины их выбора. Ищите способы упрощения структуры файлов и папок. Помните, что такую работу можно выполнять постепенно — не нужно полностью переписывать приложение самостоятельно. Используйте кодревью для обсуждения с командой лучших практик и подходов, с которыми вы сталкивались.

Не переусердствуйте с DRY

Вот предостережение о принципе DRY, которое возникло в обсуждении с Итаном Брауном.

> Раньше я был ярым сторонником DRY, но со временем смягчил свою позицию. На практике у DRY есть серьезные недостатки.
>
> 1. Добавление слоев абстракции может сократить объем кода. Ценой будет снижение ясности — вы можете получить код, который внешне делает что-то простое, но для понимания его работы приходится «продираться» через полдесятка функций. Несмотря ни на что, иногда дублирование действительно является лучшим вариантом.
> 2. Время и усилия, потраченные на создание красивой абстракции, могут не окупиться — либо потому, что вы абстрагируете что-то, что используется всего несколько раз (и не расширяется со временем), либо потому, что параметризация сделана неверно.

3. Слишком раннее введение абстракции без тщательного изучения паттернов может привести к избыточной или неподходящей абстракции. Каюсь, я и сам грешен. Иногда я чересчур увлекаюсь рисованием у себя в голове величественного каскада абстракций, а потом, уже позже, понимаю, что был вариант получше.

Я по-прежнему считаю, что DRY — это полезный принцип, но сейчас также учитываю, что там есть свои нюансы.

Изучите различные подходы к реализации приложения. В бэкенде вы встретите микросервисы, монолиты, SQL и NoSQL — все эти варианты допустимы, если их использование обоснованно. Эти знания расширят ваш инструментарий, поскольку вы познакомитесь с решениями, которые другие разработчики уже внедрили в команде. По поводу каких-то решений у вас сложится свое мнение, а к чему-то придется адаптироваться. Главное, сохранять открытость при изучении кодовой базы и понимании рабочих процессов команды.

Добавление тестов

Некоторые существующие приложения создавались таким образом, что тестовое покрытие не стояло в приоритете. При работе над новыми функциями и рефакторинге не бойтесь добавлять тесты — это ваш вклад в улучшение процессов разработки. Если вы заметите низкий уровень покрытия тестами, проявите инициативу: предложите команде постепенно его повысить. Написание тестов для существующего кода — отличный способ разобраться в логике работы системы и пользовательских сценариях.

Если в проекте уже высокий уровень покрытия тестами, проверьте возможность внедрения сквозных тестов, если они еще не используются. Вы можете создавать тикеты на добавление тестов для существующей функциональности и параллельно оценивать уровень документации приложения. Совместная работа с продуктовой командой даст вам понимание процессов взаимодействия между отделами и ожиданий владельцев продукта. Мы рассматривали вопросы тестирования в нескольких главах этой книги, поэтому используйте эти знания, чтобы предложить улучшения или новые метрики.

Важно соблюдать баланс при добавлении тестов в существующий проект, так как этот процесс может оказаться слишком дорогим. В некоторых случаях против написания тестов могут быть веские аргументы — они требуют времени на разработку и поддержку при изменениях кода. Поэтому согласуйте с командой обоснованность тестирования в вашем конкретном случае.

Знакомство с различными типами оповещений

Вы можете получать email-оповещения при возникновении ошибок в приложениях и сначала не понимать их. Уделите время изучению системы оповещений в команде:

разберитесь, как они работают, откуда берутся ошибки и где публикуются оповещения. Убедитесь, что у вас есть доступ к инструментам логирования и серверам, где хранятся лог-файлы.

По мере освоения вами приложения от вас будут ожидать анализа ошибок и предупреждений. Изучите существующие логи, соберите статистику по частым ошибкам и определите, в какой части стека они возникают. Фиксируйте эти наблюдения в заметках, чтобы систематизировать процесс обработки ошибок. Когда возникает ошибка, попробуйте разобрать ее в паре с другим разработчиком — это поможет понять методику поиска первопричины и познакомит с системами, с которыми вы редко работаете. Важно понимать, что не все оповещения связаны с кодом. Вы научитесь определять, какие из них релевантны вашей команде, как организована коммуникация и процесс принятия решений.

Работая как с новыми, так и с ранее разработанными приложениями, вы сможете определить предпочтительное направление профессионального роста. Получая опыт в разных областях, вы постепенно начнете тяготеть к определенным направлениям. Это момент, когда нужно принимать осознанные решения: углублять знания в конкретной сфере и выбирать оптимальный карьерный путь для текущего этапа.

Траектория вашей карьеры

На протяжении этой книги вы освоили ключевые аспекты создания полноценного фулстек-приложения с долгосрочной поддержкой. Вы научились эффективно взаимодействовать практически со всеми командами организации и поняли важность этой коммуникации. Вы проанализировали продуктовую дорожную карту и приняли участие в формировании технического видения. Вы руководили командой разработчиков в обсуждениях и в процессе реализации функций.

На данном этапе карьеры вы углубляете свои навыки, сталкиваясь с более сложными техническими задачами и беря на себя руководство менее опытными разработчиками. Такое углубление ставит многих senior-разработчиков перед выбором дальнейшего профессионального пути. Накопив знания во фронтенде, бэкенде, базах данных, специфике бизнеса и принципах командного взаимодействия, вы получаете прочную основу для любых вариантов своего развития.

Техническое направление

Некоторые разработчики предпочитают оставаться на техническом пути, переходя на роли с большей ответственностью — архитектора, staff-инженера (https://noidea.dog/staff) или ведущего инженера. Вы по-прежнему сможете писать код, но ваше влияние как наставника значительно возрастет. Погружаясь глубже в техническую сторону, вы начнете работать скорее на уровне системы, чем отдельных строк кода. Хотя тикеты будут появляться в вашей работе, основное внимание будет сосредоточено на интеграции различных частей системы.

Например, вместо реализации отдельных эндпоинтов вы будете продумывать, как они должны взаимодействовать с базой данных, какие события запускать, какие соглашения об именовании использовать в соответствии с предметной областью и какую схему данных применять. Вам предстоит создавать и документировать диаграммы, показывающие взаимосвязи всех этих компонентов, а затем представлять их команде для получения обратной связи. Вы также будете отвечать за помощь коллегам, когда те столкнутся с проблемами отладки, затрагивающими несколько частей системы и внешние сервисы.

По мере продвижения по техническому пути вы возьмете на себя ответственность за поддержку актуальности пакетов и анализ их влияния на кодовую базу. Вы будете формировать видение, объединяющее несколько команд в рамках инженерного отдела, а также участвовать в разработке и реализации технической дорожной карты. Это означает, что вы станете уделять больше времени размышлениям о дальнейшем развитии приложения и подготовке к этим изменениям. Сюда входит создание внутренних инструментов для устранения блокировок в функционировании команд разработки, отслеживание ключевых метрик (например, времени ревью пул-реквестов и точности оценок), а также понимание того, как технические задачи вписываются в общую стратегию организации.

Еще один навык, который вам предстоит развить, — это коммуникация. По мере перехода на уровень senior-разработчика то, как вы общаетесь с другими, становится все важнее. Вам нужно будет наставлять начинающих разработчиков, помогать продуктовой команде управлять рисками, когда оценки сроков оказываются неточными, а также добиваться признания заслуг других членов команды. Чем дальше вы продвигаетесь по этому пути, тем больше ваша роль сводится к тому, чтобы помогать команде эффективно и быстро справляться с тикетами. Обычно это означает, что вы будете меньше писать код и больше размышлять о том, как улучшить каждую часть системы, чтобы поддерживать продуктивность всех.

Развитие коммуникативных навыков также может привести вас к роли техлида, если вас больше интересует лидерская сторона технического пути. Такая роль обычно подразумевает больше проектного менеджмента: вы работаете с командой, чтобы соблюдать сроки, и с продуктовой командой, чтобы глубоко понимать цели и объяснять ей технические ограничения. На этом этапе вы также сотрудничаете с инженерным менеджером, чтобы поддерживать или улучшать культуру команды. Как техлид, вы можете иногда подключаться к решению конкретных тикетов, если это необходимо для соблюдения дедлайнов.

Если вы решите развиваться по техническому пути, вам будет полезно следить за трендами отрасли, чтобы помогать команде внедрять лучшие практики. Это также даст вам представление о новых инструментах, которые могут решить проблемы команды. Уделите время углублению знаний о структуре организации — понимание взаимосвязей между командами и отделами поможет быстрее находить нужных людей при возникновении вопросов.

Путь в менеджмент

Альтернативный путь — переход в управление, включая роли инженерного менеджера, помощника руководителя, руководителя и вице-президента. Эти роли обычно полностью выводят вас из задач написания кода и требуют сосредоточиться на коучинге, развитии команды и стратегических аспектах инженерного отдела. Хотя в некоторых организациях вы можете иногда писать код, это зависит от масштаба компании и специфики роли. Основные задачи включают регулярные встречи с членами команды для обсуждения их дел и целей по работе.

Вы будете формировать культуру инженерного отдела и карьерные лестницы, по которым могут развиваться разработчики. В ваши обязанности войдет анализ затрат команд на инструменты и поиск более эффективных альтернатив. Основная часть работы будет заключаться в том, чтобы помогать членам команды профессионально расти, — вы будете активно слушать их на индивидуальных встречах и фиксировать их потребности. Вам также придется давать качественную обратную связь об их работе — это важно.

Когда сотрудники демонстрируют успехи, признавайте их заслуги как на личных встречах, так и публично, — это мотивирует их и обеспечивает заслуженное признание, что необходимо для продвижения по карьерной лестнице. В то же время критика тоже важна — следует конструктивно указывать на зоны роста: только так члены команды смогут развиваться.

Если члены команды выражают обеспокоенность по поводу чьей-то работы, важно оперативно сообщить сотруднику об этом. Лучше предупредить его задолго до разбора, чтобы он имел достаточно времени на внесение изменений. Для вас это отличная возможность побыть наставником, предложить способы улучшения и предоставить ресурсы для развития необходимых навыков. В ходе индивидуальных встреч обязательно запрашивайте обратную связь о том, как вы можете лучше поддерживать команду. Поскольку получить конструктивную критику от коллег бывает сложно, подавайте личный пример, первым открыто обсуждая зоны своего роста. Понимание потребностей команды — ключевая часть вашей работы, поэтому основной способ профессионального развития — это регулярный сбор фидбека.

Exit-интервью (собеседование с сотрудником в связи с его увольнением) и анонимные опросы — эффективные источники критической обратной связи. Такие инструменты особенно полезны, поскольку снимают психологическое давление в ситуациях, когда члены команды стесняются напрямую указывать на ваши слабые места.

При переходе на позицию руководителя менеджеров важно создать документацию, которая поможет проводить эффективные индивидуальные встречи и вести сложные разговоры с подчиненными. Эта роль уникальна тем, что вам, скорее всего, больше не придется работать с кодом напрямую, но ваши материалы станут основой для других менеджеров, которые вам подотчетны. Следует документировать ключевые аспекты: примеры вопросов для собеседований, темы для обсуждения

тет-а-тет, методы вовлечения сотрудников в корпоративную культуру и стратегии профессионального роста команды. Все это способствует развитию их управленческих навыков.

На этом уровне вы также будете чаще взаимодействовать с продуктовой командой и стейкхолдерами, например с вице-президентами других департаментов или топменеджментом. Эти встречи помогут понять бизнес-стратегию компании и роль ваших команд в ее реализации. Вы будете напрямую участвовать в формировании продуктовой дорожной карты и согласовании с ними технического плана развития. Кроме того, в ваши обязанности войдут принятие решений о найме новых сотрудников и оценка необходимости расширения команды для достижения целей организации.

Коммуникация становится ключевым навыком для руководителя, поскольку практически вся ваша работа будет сводиться именно к ней. Вы больше не будете работать с тикетами, а ваше участие в спринтах и повседневных задачах разработки существенно сократится. Основное время будет посвящено встречам и документированию их результатов таким образом, чтобы это преобразовывалось в конкретные задачи для ваших команд. Ваша роль — стать щитом для команд, защищая их от избыточных встреч, где нужно что-то уточнить или прояснить.

Вам необходимо научиться балансировать между личными взглядами и организационными решениями, четко расставляя приоритеты. Важно уметь аргументированно возражать против решений, которые могут негативно повлиять на команды. Часть управленческой работы — это продвижение интересов ваших подчиненных и обеспечение учета их потребностей в стратегических решениях компании. Не бойтесь озвучивать сомнения, когда что-то кажется вам нелогичным, — это одна из самых ценных функций вашей позиции.

Если вы рассматриваете переход на руководящую должность, вот полезные материалы:

- *The Manager's Path*[1] Камиль Фурнье (Camille Fournier) (издательство O'Reilly);
- *Engineering Management for the Rest of Us* Сары Драснер (Sarah Drasner) (издательство Skill Recordings Inc.);
- *Resilient Management* Лары Хоган (Lara Hogan) (издательство A Book Apart).

Профессиональный журнал

Настоятельно рекомендую вести личные заметки о наблюдениях и решениях в разных проектах. Это станет вашим шаблоном для работы в будущем. Лучшие практики из каждой организации можно нести с собой дальше по карьерному пути. Такой подход поможет на каждом этапе — со временем вы начнете замечать повторяющиеся сценарии в разных компаниях.

---

[1] На русском зыке: Фурнье Камиль. От разработчика до руководителя. Менеджмент для IT-специалистов.

Записи из профессионального журнала

Примеры записей из моего личного журнала.

Проблема с поиском кастомного npm-пакета в пайплайне CircleCI.

Возвращалась ошибка: конкретный пакет не найден. Решение таково:

- удалить `package-lock.json`;
- переустановить все пакеты командой `npm i`;
- отправить изменения через пул-реквест в ветку с проблемным пайплайном;
- смержить обновленный `package-lock.json` и перезапустить пайплайн.

Проблема с модульным тестом, который проходит при индивидуальном запуске, но падает при выполнении со всеми остальными тестами.

Данные изменялись при каждом вызове функции. Это приводило к нестабильным результатам — тест мог проходить только в некоторых случаях. Решение включало следующие шаги:

- выявление использования метода массива, который изменял исходный массив;
- применение оператора `spread` к исходному массиву для создания его нового экземпляра;
- использование метода массива уже на этом новом экземпляре.

Минутка гордости: устранение двухлетнего бага в интерфейсе.

В одном проекте существовал трудноуловимый баг в UI из-за использования микрофронтенд-архитектуры, где минимум шесть команд отвечали за разные части функциональности, причем каждая работала по-своему. Особенно осложняло ситуацию то, что одна команда разрабатывала фронтенд на Vue, тогда как остальные использовали React. Мне удалось разобраться с многочисленными конфликтами зависимостей и обнаружить, что одна из микрофронтенд-команд принудительно устанавливала свой пакет в контейнерное приложение, чтобы избежать обновления внутренних зависимостей. Это вынудило остальные команды создавать обходные решения для данной ошибки. На устранение конфликтов между шестью приложениями ушло около двух с половиной месяцев, но это также исправило баг, существовавший годами. После завершения обновлений каждая команда смогла провести рефакторинг для повышения качества кода, а производительность всего приложения улучшилась благодаря удалению неиспользуемых пакетов и уменьшению размеров бандлов как для микрофронтенд-команд, так и для команды контейнерного приложения.

Вам стоит вести записи о ситуациях, которыми вы особенно гордитесь, и о ситуациях, где испытывали наибольшие трудности. Такие записи бесценны при собеседованиях на новые должности, поскольку вам неизбежно будут задавать ситуационные или поведенческие вопросы. Эти записи также полезны во время ежегодных обзоров в текущей роли — за полгода или год легко забыть о своих достижениях, поэтому лучше фиксировать их письменно.

Когда вы осваиваете новые технические практики, принятые в организации, добавляйте их в этот журнал. Вы удивитесь, как часто схожие проблемы встречаются в разных командах, компаниях и отраслях. Если у вас возникает ощущение, что вы уже сталкивались с подобной задачей, журнал поможет это подтвердить. Лично я многократно фиксировала сложные баги или спецификации функций, а позже использовала эти записи при повторении аналогичных сценариев в других проектах.

Этот журнал следует вести на личном компьютере, а не на рабочем. Вы никогда не знаете, когда можете потерять доступ к корпоративному устройству или когда его данные будут стерты. Помните, что это неформальный документ — он только для вас, поэтому формат может быть любым. Во вставке на с. 427 приведен шаблон, по которому сделано большинство записей в моем профессиональном журнале. Для технических проблем в нем указывается, что произошло и что я сделала для их решения. Для ситуаций, которыми я горжусь или которые вызвали трудности, я стараюсь описать историю произошедшего и то, как все развивалось.

Переход в другие области

После долгого движения в одном направлении или смены специализации вы можете осознать необходимость кардинальных изменений и решить попробовать что-то совершенно новое. Возможно, вас всегда интересовал Data Engineering и у вас уже есть опыт работы с базами данных. Или вы захотели стать DevOps-инженером после небольшого опыта работы с инфраструктурой, который пробудил ваш интерес. Совершенно нормально освоить новую специализацию и какое-то время развиваться в этом направлении.

Если в вашей организации есть возможности, вы можете поучаствовать в различных инициативах и параллельно прокачивать свои навыки. Также можно рассмотреть переход на позицию ниже senior в новой области, например, стать junior-инженером данных. Ваши основные навыки разработчика никуда не денутся, а погружение в другую технологическую сферу может повысить креативность, поскольку вы увидите процессы под новым углом. Это подходящее время для прохождения сертификации или работы над небольшими функциональными задачами, которые нужны вашей организации. Однако у такого исследования есть и обратная сторона: учитывая постоянные изменения в технологических стеках, вы можете отстать от последних тенденций в своей основной специализации.

Независимо от выбранного вами пути, вам очень поможет прохождение какого-либо обучения по управлению. Узнайте, есть ли в вашей организации рекомендуемые программы или доступ к ним. Мне помогли освоить управление несколько книг, хотя некоторые из них не совсем очевидны для такой цели:

- *How to Win Friends and Influence People* Дейла Карнеги (Dale Carnegie) (Simon & Schuster);
- *Thinkertoys: A Handbook of Creative-Thinking Techniques* Майкла Микалко (Michael Michalko) (Ten Speed Press);
- *Spark: How to Lead Yourself and Others to Greater Success* Энджи Морган, Кортни Линч и Шона Линча (Angie Morgan, Courtney Lynch, and Sean Lynch) (Houghton Mifflin Harcourt);
- *Wherever You Go, There You Are: Mindfulness Meditation in Everyday* Life Джона Кабат-Зинна (Jon Kabat-Zinn) (Hachette Books)[1].

Вы всегда можете поискать себя в разных специализациях. Например, попробуете работать с данными — и поймете, что это не ваше. Тогда можно вернуться к разработке приложений, пока не найдете то, что вам по-настоящему интересно. А возможно, вы поймете, что от добра добра не ищут — вас полностью устраивает текущая роль. Не все стремятся как можно быстрее карабкаться по карьерной лестнице и получать повышения — и это абсолютно нормально. Это ваша карьера, и правил здесь не существует. Делайте то, что лучше для вас и вашей жизни, ведь только вы проживаете ее.

Заключение

Поздравляю! Вы дошли до конца этой книги. К этому моменту вы освоили все необходимое для создания поддерживаемого фулстек-приложения — и на фронтенде, и на бэкенде. Вы научились оценивать инструменты, выстраивать коммуникацию между командами, фиксировать технические решения и поддерживать коллег, когда это требуется. Вы также узнали о менее очевидных, но крайне важных навыках senior-разработчика: например, о том, как писать понятную документацию и грамотно применять свой опыт в работе.

Я искренне надеюсь, что эта книга оказалась для вас полезной и вы будете обращаться к ней, когда столкнетесь с трудностями в процессе разработки. Если какие-то разделы помогли вам — буду рада услышать о вашем опыте! А если вам кажется, что где-то я ошиблась или упустила важные моменты, — тоже пишите. Хотя эта книга не охватывает все возможные сценарии, в ней рассмотрены многие стандартные ситуации. Помните: с годами и с развитием карьеры перед вами всегда будут открываться новые знания. Мы, разработчики, никогда не знаем всего, поэтому сохраняйте открытость и скромность. Это поможет вам пройти долгий путь.

---

[1] На русском языке: Карнеги Дейл. Как завоевывать друзей и оказывать влияние на людей; Микалко Майкл. Игры для разума. Тренинг креативного мышления; Кабат-Зинн Джон. Куда бы ты ни шел, ты уже там: осознанная медитация в повседневной жизни.

Об авторе

Милесия Макгрегор — ведущий инженер-разработчик, работала с TypeScript, Angular, React, Node, Python, SQL, AWS и многими другими инструментами для создания веб-приложений. Имеет степень магистра в области машиностроения и аэрокосмической инженерии, автор научных публикаций в области машинного обучения и робототехники. Милесия выступала на международных технологических конференциях и работала на позициях от фронтенд-разработчика до заместителя директора по разработке. Она публикует статьи, посвященные различным аспектам разработки программного обеспечения, в ряде изданий, включая freeCodeCamp. Свободное время проводит с мужем и собаками, занимается боевыми искусствами и готовится к получению лицензии пилота.

Иллюстрация на обложке

На обложке книги «Фулстек JavaScript: Секреты, которые должен знать каждый мидл» изображен австралийский совиный лягушкорот (tawny frogmouth, Podargus strigoides) — вид птиц из семейства лягушкоротов, обитающий в лесах и редколесьях Австралии и Тасмании. Несмотря на внешнее сходство с совами, совиные лягушкороты находятся в более близком родстве с козодоями. Они отличаются широким, «лягушачьим» клювом, крупными желтыми глазами и мягким пестрым оперением коричневых и серых оттенков, обеспечивающим отличную маскировку на фоне древесной коры.

Совиные лягушкороты ведут преимущественно ночной образ жизни; это хищники, питающиеся различными насекомыми, пауками, червями, слизнями, улитками, мелкими млекопитающими, рептилиями и лягушками. Они известны своей исключительной способностью к маскировке и бесшумным полетом, что делает их весьма эффективными охотниками.

В настоящее время австралийский совиный лягушкорот имеет охранный статус «вызывающие наименьшие опасения» по классификации Международного союза охраны природы (IUCN). Однако утрата среды обитания вследствие вырубки лесов и урбанизации представляет потенциальную угрозу для его популяции. Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой исчезновения; все они важны для мира природы.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из Brehms Tierleben. Дизайн серии разработан Эди Фридман, Элли Фолькхаузен и Карен Монтгомери.

Милесия МакГрегор

Фулстек JavaScript: Секреты, которые должен знать каждый мидл

Перевел с английского Е. Бартов

Научный редактор В. Краснолуцкий

Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 04.2026. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 27.03.26. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 700. Заказ 0000.

Read IT Club

Комьюнити рецензентов и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы на русском языке. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.