O'REILLY®

Микросервисы

от архитектуры до релиза

пошаговое руководство

Ронни Митра
Иракли Надареишвили

Microservices: Up and Running

A Step-by-Step Guide to Building a Microservices Architecture

Ronnie Mitra and Irakli Nadareishvili

Beijing • Boston • Farnham • Sebastopol • Tokyo

Микросервисы
от архитектуры до релиза

пошаговое руководство

Ронни Митра, Иракли Надареишвили

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018
УДК 004.457
М62

Митра Р., Надареишвили И.

М62 Микросервисы. От архитектуры до релиза. — СПб.: Питер, 2023. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1884-7

Микросервисная архитектура обеспечивает высокую скорость изменений и хорошую масштабируемость, а также позволяет создавать чистые эволюционирующие системы. Но реализовать свою первую микросервисную архитектуру непросто. Как сделать выбор из множества вариантов и обучить свою команду всем техническим деталям, чтобы максимально увеличить шансы на успех? В этой книге авторы, Ронни Митра и Иракли Надареишвили, предоставили пошаговое руководство для построения эффективной архитектуры микросервисов. Архитекторы и инженеры пройдут путь внедрения, основанный на методах и архитектурах, доказавших свою эффективность для микросервисных систем. Вы создадите операционную модель, проект микросервиса, инфраструктурную основу и два работающих микросервиса, а затем соедините эти компоненты в одну реализацию. Для любого, перед кем стоит задача создания микросервисов, руководство станет бесценным источником знаний.

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

ББК 32.973.2-018
УДК 004.457

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. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. 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.

ISBN 978-1492075455 англ
Authorized Russian translation of the English edition of Microservices: Up and Running, ISBN 9781492075455 © 2021 Mitra Pandey Consulting, Ltd. and Irakli Nadareishvili.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same
ISBN 978-5-4461-1884-7
© Перевод на русский язык ООО «Прогресс книга», 2023
© Издание на русском языке, оформление ООО «Прогресс книга», 2023
© Серия «Бестселлеры O'Reilly», 2023

Краткое содержание

Предисловие ... 14

Глава 1. Навстречу архитектуре микросервисов ... 18

Глава 2. Разработка операционной модели микросервисов ... 33

Глава 3. Разработка микросервисов: процесс SEED(S) ... 56

Глава 4. Выбор оптимального размера микросервисов: определение границ сервисов ... 81

Глава 5. Работа с данными ... 101

Глава 6. Создание конвейера инфраструктуры ... 125

Глава 7. Создание инфраструктуры микросервисов ... 168

Глава 8. Рабочая область разработчика ... 213

Глава 9. Разработка микросервисов ... 232

Глава 10. Выпуск микросервисов ... 268

Глава 11. Управление изменениями ... 302

Глава 12. Конец путешествия (и новое начало) ... 322

Об авторах ... 332

Иллюстрация на обложке ... 333

Оглавление

Предисловие ... 14
- Для кого эта книга ... 14
- Что вам понадобится ... 15
- Условные обозначения ... 15
- Использование примеров кода ... 16
- Благодарности ... 17
- От издательства ... 17

Глава 1. Навстречу архитектуре микросервисов ... 18
- Что такое микросервисы ... 19
- Сокращение затрат на координацию ... 21
  - Проблема затрат на координацию ... 22
  - Сложные компромиссы ... 24
- Обучение на практике ... 25
  - Модель микросервисов «От архитектуры до релиза» ... 26
- Решения, решения... ... 28
  - Создание упрощенной записи архитектурного решения ... 29
- Резюме ... 31

Глава 2. Разработка операционной модели микросервисов ... 33
- Почему команды и люди важны ... 34
  - Размер команды ... 35
  - Мастерство команды ... 37
  - Взаимодействие между командами ... 38

Введение в Team Topologies 40
Тип команды 41
Режимы взаимодействия 42
Разработка топологии команды по работе с микросервисами 43
Создание команды разработки системы 44
Подготовка шаблона команды по созданию микросервисов 46
Команды разработки платформы 49
Команды поддержки и разработки сложных подсистем 51
Команды потребителей 52
Резюме 54

Глава 3. Разработка микросервисов: процесс SEED(S) 56
Семь основных этапов эволюционного проектирования сервисов: метод SEED(S) 57
Идентификация участников 58
Примеры участников в нашем учебном проекте 60
Определение действий, выполняемых участниками 61
Использование формата истории заданий для фиксации JTBD 63
Примеры JTBD в нашем проекте 64
Выявление шаблонов взаимодействия с помощью диаграмм последовательностей 65
Выделение действий и запросов из JTBD 68
Примеры запросов и действий для нашего проекта 69
Описание каждого запроса и действия в виде спецификации с использованием открытого стандарта 71
Пример OAS для действия в нашем проекте 72
Получение обратной связи по спецификации API 76
Реализация микросервисов 77
Микросервисы и API 77
Резюме 80

Глава 4. Выбор оптимального размера микросервисов: определение границ сервисов ... 81

Почему границы имеют значение, когда они имеют значение и как их найти ... 82

Предметно-ориентированное проектирование и границы микросервисов ... 84

Составление карты контекста ... 87

Синхронные и асинхронные интеграции ... 90

Агрегаты в DDD ... 91

Введение в Event Storming ... 92

Процесс Event Storming ... 94

Представляем универсальную формулу определения размера ... 99

Резюме ... 100

Глава 5. Работа с данными ... 101

Возможность независимого развертывания и обмена данными ... 101

Микросервисы владеют своими данными ... 103

Владение данными не должно приводить к резкому увеличению количества кластеров базы данных ... 104

Владение данными и шаблон делегирования данных ... 105

Дублирование данных для решения проблемы независимости ... 108

Распределенные транзакции и защита от сбоев ... 109

Event Sourcing и CQRS ... 112

Event Sourcing ... 112

Повышение производительности с помощью скользящих снимков ... 118

Хранилище событий ... 119

Разделение ответственности на команды и запросы (CQRS) ... 121

Event Sourcing и CQRS за пределами микросервисов ... 122

Резюме ... 124

Глава 6. Создание конвейера инфраструктуры ... 125
Принципы и практики DevOps ... 127
Неизменяемая инфраструктура ... 127
Инфраструктура как код ... 129
Непрерывная интеграция и непрерывное развертывание ... 131
Настройка среды IaC ... 133
Настройка GitHub ... 133
Установка Terraform ... 134
Настройка Amazon Web Services ... 135
Настройка операционной учетной записи AWS ... 136
Настройка AWS CLI ... 140
Настройка разрешений AWS ... 141
Создание серверной части S3 для Terraform ... 145
Создание конвейера IaC ... 147
Создание репозитория «песочницы» ... 148
Понимание Terraform ... 150
Написание кода для «песочницы» ... 151
Создание конвейера ... 154
Тестирование конвейера ... 164
Резюме ... 167

Глава 7. Создание инфраструктуры микросервисов ... 168
Компоненты инфраструктуры ... 168
Сеть ... 169
Сервис Kubernetes ... 170
Сервер развертывания GitOps ... 172
Создание инфраструктуры ... 173
Установка kubectl ... 173
Настройка репозиториев для модулей ... 174
Модуль определения сети ... 177

Модуль Kubernetes ... 192
Настройка Argo CD ... 204
Тестирование среды ... 208
Очистка инфраструктуры ... 210
Резюме ... 212

Глава 8. Рабочее пространство разработчика ... 213
Стандарты программирования и настройки среды разработки ... 214
Десять рекомендаций по рабочей области для улучшения работы разработчика ... 215
Локальная настройка контейнерной среды ... 222
Установка Multipass ... 223
Вход в контейнер и отображение папок ... 225
Установка Docker ... 226
Использование локальной версии Docker: установка Cassandra ... 228
Установка Kubernetes ... 229
Резюме ... 231

Глава 9. Разработка микросервисов ... 232
Проектирование конечных точек микросервисов ... 232
Микросервис управления информацией о рейсах ... 236
Микросервис управления бронированием ... 237
Проектирование спецификации OpenAPI ... 238
Реализация данных для микросервиса ... 245
Redis для модели данных бронирования ... 245
Модель данных MySQL для микросервиса управления информацией о рейсах ... 247
Реализация кода микросервиса ... 249
Код микросервиса управления информацией о рейсах ... 250
Проверки работоспособности ... 255

Ввод второго микросервиса в проект ... 257
Подключение сервисов к зонтичному проекту ... 264
Резюме ... 267

Глава 10. Выпуск микросервисов ... 268
Настройка среды обкатки ... 269
Модуль входного шлюза ... 270
Модуль базы данных ... 271
Разветвление проекта инфраструктуры обкатки ... 272
Настройка потока среды обкатки ... 273
Редактирование кода инфраструктуры обкатки ... 275
Отправка контейнера с информацией о рейсах ... 279
Введение в Docker Hub ... 280
Настройка Docker Hub ... 281
Настройка конвейера ... 281
Развертывание контейнера с информацией о рейсах ... 285
Особенности развертывания в Kubernetes ... 286
Создание чарта Helm ... 287
Создание репозитория развертывания микросервисов ... 288
Argo CD для развертывания GitOps ... 294
Очистка ... 301
Резюме ... 301

Глава 11. Управление изменениями ... 302
Изменения в системе микросервисов ... 302
Ориентация на данные ... 303
Влияние изменений ... 304
Три шаблона развертывания ... 306
Обзор нашей архитектуры ... 308
Изменения в инфраструктуре ... 309

Изменения микросервисов 313
Изменения данных 318
Резюме 321

Глава 12. Конец путешествия (и новое начало) 322
О сложности и упрощении использования микросервисов 323
Квадрант микросервисов 325
Оценка прогресса трансформации на пути к микросервисам 327
Резюме 331

Об авторах 332

Иллюстрация на обложке 333

Каждому, кто нашел время вести хронику и поделился своим опытом. И Кайраву, который отказался помогать мне писать посвящение.

Ронни Митра

Лукасу, родившемуся вскоре после того, как мы начали работать над этой книгой. Его улыбки придали мне сил закончить этот труд в разгар глобальной пандемии. Моей жене Ане за ее поддержку и моим замечательным студентам из Университета Темпл в Филадельфии, которые любезно «протестировали» ранние версии многих материалов книги.

Иракли

Предисловие

Десять лет назад команда архитекторов программного обеспечения ввела термин *«микросервисы»*, определяя эволюционировавший стиль архитектуры программного обеспечения (ПО). С тех пор было выпущено множество курсов, видеороликов и книг, посвященных микросервисам. Фактически в 2016 году мы стали соавторами книги *Microservice Architecture* (https://learning.oreilly.com/library/view/microservice-architecture/9781491956328) — вводного руководства по микросервисам.

С момента публикации той книги у нас и у многих других было достаточно времени, чтобы накопить опыт работы с созданными нами микросервисами. Наш собственный опыт, а также беседы с другими практикующими специалистами позволили нам лучше понять реальные проблемы, с которыми сталкиваются разработчики.

Мы постарались объединить опыт практикующих специалистов в авторитетное руководство. Мы живем в эпоху, когда доступно множество прикладных советов. Но ориентироваться в этом море информации довольно сложно, как и собрать ее воедино так, чтобы извлечь максимальную пользу. В этой книге предлагается практическая модель, охватывающая организацию команды разработчиков, приемы предметного проектирования, подходы к созданию инфраструктуры, разработке и выпуску. Наша цель — дать вам единое представление о реализации микросервисов и помочь сделать решительный первый шаг на пути к внедрению.

Для кого эта книга

Мы написали эту книгу для разработчиков микросервисов. Затрагивая некоторые принципы и модели системы микросервисов, основное внимание мы уделяем практическому проектированию и разработке. Если вы архитектор или инженер микросервисов, то эта книга для вас. Но она также будет полезна читателям, которые просто хотят больше узнать о реализации микросервисов.

Что вам понадобится

Поскольку сфера применения микросервисов довольно велика, мы используем ряд различных инструментов и методов. Если вы решите опробовать примеры из книги, то установите следующие инструменты и платформы или оформите подписку на них:

- Docker;
- Redis;
- MySQL;
- GitHub;
- GitHub Actions;
- Terraform;
- Amazon Web Services;
- kubectl;
- Helm;
- Argo CD.

Инструкции о том, где и как получить эти инструменты, мы представим в соответствующих разделах.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины.

`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, имена файлов и их расширений, названия каталогов.

`Моноширинный жирный шрифт`

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

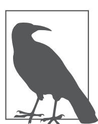
`Моноширинный курсив`

Показывает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

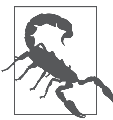
Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, названий клавиш и их сочетаний, элементов интерфейса.

Этот рисунок указывает на совет или предложение.



Этот рисунок указывает на общее замечание.



Этот рисунок указывает на предупреждение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) можно скачать здесь: https://oreil.ly/MicroservicesUpandRunning.

Если у вас есть технический вопрос или проблема, касающиеся примеров кода, то, пожалуйста, напишите на электронную почту bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Мы хотели бы поблагодарить наших редакторов Мелиссу Поттер и Дебору Бэкер, а также команду O'Reilly, без которой мы никогда не закончили бы книгу. Спасибо Питу Ходгсону, Крису О'Деллу, Лоринде Брэндон, Джей Пи Моргенталю, Майку Андерсену и Дэвиду Батленду за вдохновение и содержательные отзывы. И наконец, мы хотели бы поблагодарить компании Capital One и Publicis Sapient за поддержку, которую они оказали, позволив нам воплотить нашу идею в жизнь.

От издательства

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

Мы будем рады узнать ваше мнение!

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

ГЛАВА 1

Навстречу архитектуре микросервисов

Цель книги — помочь вам создать рабочую архитектуру микросервисов. Здесь вы найдете надежные и проверенные советы по созданию ПО. Они основаны на опыте реальных практикующих специалистов, как на примерах успешных реализаций, так и на тех, которые могли быть реализованы лучше. Мы превратили эти уроки в модель, которая, как мы надеемся, поможет вам быстрее освоиться с вашей системой.

В последнее время популярность создания программного обеспечения в виде микросервисов резко возросла. В начале 2010-х годов термин *«микросервисы»* был введен для описания нового стиля архитектуры ПО. Приложения, реализованные в этом стиле, создаются с помощью небольших независимых компонентов, которые работают вместе. С тех пор темпы внедрения стиля микросервисов резко возросли. Стартапы, крупные и мелкие компании изучают и внедряют микросервисные архитектуры. Растущая экосистема инструментов, сервисов и решений в данной области свидетельствует о ее широкой популярности. Когда мы писали эту книгу, специалисты Allied Market Research (https://oreil.ly/cugsz) предсказали, что мировой рынок микросервисных архитектур вырастет до 8,07 млрд долларов США в 2026 году с нынешних 2,07 млрд долларов США. Такие цифры указывают на большой интерес к микросервисам и их широкое внедрение.

Для многих реализация микросервисных архитектур оказалась нелегкой задачей. Дело в том, что внедрить систему микросервисов непросто. Заставить множество независимых частей работать вместе — куда сложнее, чем может показаться. Затраты на управление, сопровождение, поддержку и тестирование в такой системе суммируются. В крупных масштабах эти затраты могут стать непомерно высокими. Если вы не будете осторожны, то сложность управления системой может привести к тому, что микросервисы окажутся плохой затеей.

Но преимущества микросервисов оправдывают риски. Хорошо реализованные микросервисы позволяют развивать программное обеспечение быстрее и безопаснее, если рассматривать достаточно масштабную систему. Скорость и безопасность изменений дают бо́льшую гибкость, что способствует процветанию и вашего бизнеса, и вашей организации.

Чтобы раскрыть все эти преимущества, нужно создать правильную архитектуру для поддержки сервисов. Важно снизить системные затраты, не снижая ценности независимых сервисов. Чтобы создать такую архитектуру, нужно заранее принять важные решения, касающиеся методов, процессов, команд, технологий и инструментов. Все это должно работать слаженно, чтобы сформировать новое оптимизированное целое.

Хороший способ построить подобную систему — эволюционное развитие. Для начала можно принять несколько простых решений и учиться и расти по ходу дела. Фактически большинство пионеров в этой области пришли к микросервисам благодаря многочисленным экспериментам. Они не ставили перед собой цель создать приложение на основе микросервисов. Они получили их в результате непрерывного процесса оптимизации и совершенствования.

Чтобы начать с нуля и двигаться к цели шаг за шагом, нужно время. Но у вас есть ценное преимущество — возможность использовать опыт ваших предшественников, чтобы помочь себе быстрее выстроить свою систему. Начните ее создание с подбора шаблонов, методов и инструментов, сочетание которых привело к успеху. Затем оптимизируйте систему в соответствии с целями и ограничениями вашей организации.

В этой книге мы описали решения, формирующие прочную основу для создания микросервисов. Но прежде чем углубиться в детали, рассмотрим важный вопрос: что именно подразумевается под «микросервисами»?

Что такое микросервисы

Не существует общепринятого официального, канонического определения *микросервисов*. Хорошей отправной точкой является статья Джеймса Льюиса и Мартина Фаулера о микросервисах 2014 года (https://oreil.ly/guhCP). Авторы описывают микросервисы как:

> «подход к разработке единого приложения в виде набора небольших сервисов, работающих в отдельных процессах и взаимодействующих с применением упрощенных механизмов. [...] построенных вокруг бизнес-потребностей и развертываемых независимо с помощью полностью автоматизированного механизма».

Суть статьи Льюиса и Фаулера — в описании набора из девяти характеристик, которыми обладают микросервисы. Список начинается с основной микросервисной характеристики: *компонентное представление с помощью сервисов*, что означает разбиение приложения на более мелкие сервисы. От него авторы переходят к широкому спектру возможностей. Они обосновывают необходимость организационного и управленческого проектирования с учетом особенностей *организации, связанных с бизнес-потребностями* и *децентрализованным управлением*. Авторы намекают на DevOps и методы гибкой разработки, когда внедряют *автоматизацию инфраструктуры* и *продукты, а не проекты*. В дополнение авторы определяют несколько ключевых принципов архитектуры, таких как *умные конечные точки и тупые каналы* (smart endpoints and dumb pipes), *проектирование с учетом сбоев* и *эволюционное проектирование*.

Каждая из этих характеристик заслуживает внимания, и мы рекомендуем вам прочитать их статью, если вы еще этого не сделали. Вместе эти характеристики образуют целостное решение с очень большим набором задач, куда включаются технологии, инфраструктура, инжиниринг, внедрение, управление, структура команды и культура.

Для сравнения предлагаем еще одно определение микросервисов из книги *Microservice Architecture*, написанной Иракли Надареишвили, Ронни Митрой, Мэттом Макларти и Майком Амундсеном (O'Reilly) (https://learning.oreilly.com/library/view/microservice-architecture/9781491956328).

> «*Микросервис* — это независимо развертываемый компонент с ограниченной областью действия, поддерживающий взаимодействия посредством обмена сообщениями. *Микросервисная архитектура* — это стиль проектирования высокоавтоматизированных, эволюционирующих программных систем, состоящих из микросервисов, ориентированных на потребности».

Оно похоже на определение Льюиса и Фаулера, но здесь особое внимание уделяется ограниченным областям, функциональной совместимости и взаимодействиям посредством сообщений. В нем также проводится различие между микросервисами и архитектурой, которая их обеспечивает.

Это всего лишь два примера из множества определений микросервисов. Большинство определений в целом схожи, но каждое из них немного отличается по своей направленности.

В мире технологий названия и определения важны, поскольку позволяют нам простым языком объяснить сложные понятия. В этом случае слово «микросервисы» помогает нам описать *стиль* архитектуры ПО, обладающий тремя основными конструктивными чертами.

1. Архитектура приложения в основном состоит из машинно-вызываемых «сервисов», доступных в сети.

2. Размеры (или границы) сервисов — важный критерий проектирования, на который влияют факторы времени выполнения, времени разработки и персонала.
3. Программная система, организация и способ работы оптимизированы для достижения цели.

Это довольно общий набор конструктивных черт. Например, в нем не упоминаются организационные стили, конкретные инструменты или архитектурные принципы, которые следует использовать. Не определены и какие-либо формальные шаблоны или практики. Но он дает нам достаточно характеристик, чтобы идентифицировать систему микросервисов при встрече с ней.

Правда в том, что практически любую систему на базе API можно назвать микросервисной архитектурой, если очень постараться. Основное внимание должно быть сосредоточено на цели вашей системы. Мы думаем, что вопрос о том, зачем создаются микросервисы, гораздо важнее, чем вопрос о том, что они собой представляют. Несмотря на то что микросервисы обладают множеством потенциальных преимуществ, мы считаем, что лучшая причина создавать программное обеспечение с применением этого подхода — снижение затрат на координацию (взаимодействие).

Сокращение затрат на координацию

Компании по всему миру с успехом внедряют микросервисные архитектуры. Почти все практикующие специалисты, с которыми мы беседовали, сообщали об увеличении скорости доставки программного обеспечения. Мы считаем, что улучшение обусловлено фундаментальным преимуществом микросервисов: снижением затрат на координацию.

Следует отметить, что существует множество способов увеличить скорость разработки ПО, и организация программного обеспечения в виде микросервисов — лишь один из вариантов. Например, можно быстро создать систему, выполнив работу поверхностно и накопив «технический долг» (https://oreil.ly/PBMHU), с которым предполагается разобраться позже. Или же можно меньше внимания уделить стабильности и безопасности и просто выпустить свой продукт в мир. В отдельных ситуациях и для некоторых предприятий это разумные подходы.

Но в системах, разрабатываемых, в частности, для финансового, медицинского и государственного секторов, непозволительно ради повышения скорости идти на компромисс с безопасностью. И все же рынок требует от этих отраслей более высокой скорости, как и от любых других. Именно здесь микросервисы могут проявить себя во всей красе. Они предлагают архитектурный подход, позволяющий увеличить скорость разработки без ущерба для безопасности. И дают возможность делать это в широком масштабе.

Проблема затрат на координацию

Создание сложного программного обеспечения — тяжелая работа. В кино блестящий программист может за пару бессонных ночей героически создать продукт, способный изменить мир. В реальной жизни для получения качественного результата требуется много людей и уйма времени. Несколько команд, работающих над сложным проектом, обычно реализуют разные части общей системы, следуя своим планам и совершая шаги независимо. Периодически эти части необходимо интегрировать, чтобы устранить зависимости, для чего независимые команды должны координировать свою работу (рис. 1.1).

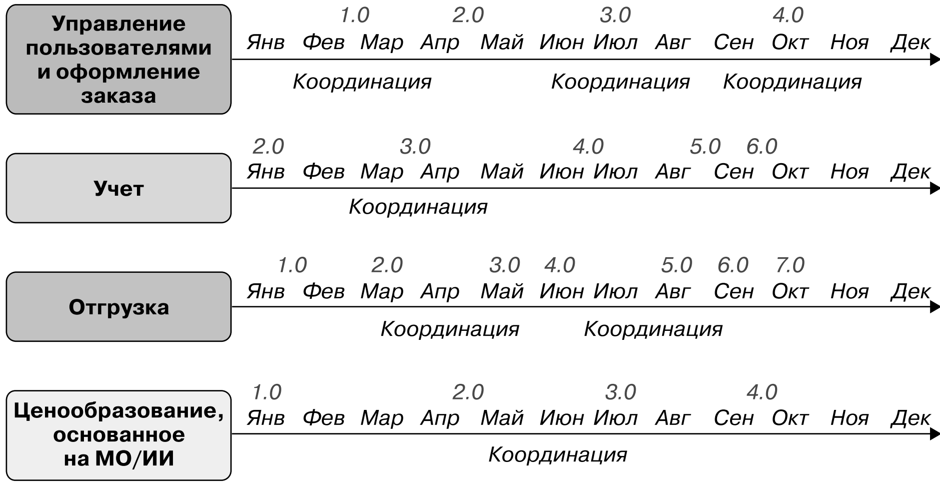


Рис. 1.1. Примерная временная шкала сложного проекта с координационными точками соприкосновения

Представьте, что Джейн — руководитель группы, отвечающей за бухгалтерский учет. Ее команда только что закончила спринт и зависит от компонента, который разрабатывается командой, отвечающей за модуль отгрузки, во главе с Тайроном. Поскольку их проектные планы независимы, вполне возможно, что его команда на самом деле не закончила реализацию необходимого компонента. На данный момент у Джейн есть два варианта: либо дожидаться готовности компонента (отдавая приоритет безопасности и жертвуя скоростью) и провести надлежащее интеграционное тестирование, либо положиться на согласованный контракт взаимодействий между ее компонентом и компонентом Тайрона, предполагая, что его команда выпустит обновленную версию компонента точно в срок. Выбрав второй вариант, Джейн сможет работать без остановки, увеличивая скорость работы своей команды, но потенциально

ставя под угрозу общую безопасность системы, поскольку интеграционное тестирование не проводилось на самом раннем возможном этапе и было сделано предположение об «удачном пути»[1].

Любой руководитель в организации со множеством задействованных команд регулярно сталкивается с этим выбором: игнорировать затраты на координацию и сохранить динамику или признать необходимость координации и замедлиться. Как правило, мы выбираем тот или иной вариант, интуитивно оценивая соотношение риска и выгоды. Но в целом в достаточно сложной системе, когда такой выбор необходимо делать часто, возникает очень заметное противоречие между скоростью и безопасностью.

Поскольку нас беспокоят затраты на координацию, то было бы неплохо иметь систему, специально разработанную так, чтобы *минимизировать* их. Чтобы вместо выбора того или иного пути команды вообще не сталкивались с необходимостью выбора. Хотелось бы иметь проект, требующий минимум координации между автономными командами, работающими над небольшими изолированными задачами. Именно это предлагает микросервисная архитектура.

Важно понимать, что работа по созданию хороших микросервисов направлена на минимизацию координации. Создавать сложные распределенные системы, такие как микросервисные архитектуры, непросто, и в моменты сомнений мы всегда должны спрашивать себя: «Решение, с которым я сталкиваюсь, снизит затраты на координацию моих команд или нет?» Правильный ответ будет гораздо очевиднее, если мы рассмотрим решения с точки зрения затрат на координацию.

В конечном счете микросервисы стали популярными, потому что помогают бизнесу добиться успеха. Современные организации испытывают невероятное давление, которое требует чаще и быстрее адаптироваться, меняться и совершенствоваться. Важно инвестировать в технологическую архитектуру, целенаправленно разработанную для изменения скорости и безопасности в масштабах крупной организации. Микросервисы позволяют компаниям, работающим в сложных областях, быть такими же гибкими, как более простые и небольшие компании, и продолжать использовать мощь и возможности своего фактического размера. Это невероятно привлекательно, и рост количества их внедрений доказывает это, однако преимущества не даются бесплатно. Требуется много и сосредоточенно работать и принимать важные решения, чтобы создать микросервисную архитектуру, которая может раскрыть эту ценность.

[1] От англ. happy path — сценарий по умолчанию, в котором отсутствуют исключительные ситуации или ошибки. — *Примеч. пер.*

Сложные компромиссы

Одно из самых больших препятствий, с которыми сталкиваются начинающие разработчики микросервисов, — огромная разветвленность системы. На первых порах можно сосредоточиться на создании небольших ограниченных сервисов. Но очень скоро придется создавать правильную инфраструктуру, модели данных, фреймворки, командные модели и процессы для их поддержки. Это большая область, которую нужно охватить, и решение всего спектра вопросов может привести к особым проблемам. Ниже описаны три серьезные проблемы проектирования, с которыми обычно сталкиваются архитекторы и инженеры микросервисов.

- *Длительные циклы обратной связи.* Одна из больших проблем — сложность оценки эффективности решений в системе микросервисов. Решения, принимаемые сегодня, могут повлечь проблемы, которые проявятся гораздо позже. Например, на старте вы можете решить использовать общую библиотеку, чтобы упростить реализацию взаимодействий между сервисами. Но со временем может выясниться, что поддержание ее в актуальном состоянии во всех микросервисах и командах оказывается огромной проблемой. Суть в том, что трудно оценить, как повлияет принимаемое вами решение, пока не возникнут сложности. Это затрудняет оценку и выбор вариантов.
- *Слишком много движущихся частей.* Система микросервисов — это сложная адаптивная система. Каждая часть системы некоторым образом влияет на другие ее части. Когда все они собираются воедино, возникает новое, непредсказуемое поведение системы. Если вы когда-либо внедряли новый инструмент или процесс в организации, то, вероятно, сталкивались с этим. Одни команды реагируют на новые стимулы и немедленно меняются, другим нужны помощь и поддержка, чтобы адаптироваться. Однако, несмотря ни на что, почти всегда приходится сталкиваться с последствиями работы людей и принятых ими решений. Например, команды, внедряющие инструменты контейнеризации Docker, неизбежно адаптируют свой цикл разработки и выпуска, приводя его в соответствие с принятой моделью развертывания контейнеров. Иногда эти последствия можно предусмотреть, но часто приходится иметь дело с непредвиденными последствиями вносимых изменений. Это затрудняет проектирование систем микросервисов. Трудно предсказать конкретные последствия вносимых изменений, поэтому, внедряя новую архитектурную модель, мы рискуем принести больше вреда, чем пользы.
- *Аналитический паралич.* Сложность проектируемой системы вкупе с длительными циклами обратной связи обнажает трудности построения микро-

сервисной архитектуры. Решения, которые приходится принимать, имеют большое влияние и трудно поддаются оценке. Это может привести к бесконечным спекуляциям, обсуждениям и оценке архитектурных решений из-за страха создать неправильную систему. Вместо того чтобы создать систему, способную удовлетворить требования бизнеса, мы оказываемся в подвешенном состоянии, пытаясь смоделировать бесконечные последствия нашего выбора. Это состояние широко известно как *аналитический паралич*. Мешает также наличие в Интернете множества страшных историй, советов в стиле «наклейки на бампер» и противоречивых рекомендаций по созданию микросервисных архитектур.

В конечном счете реальная сложность построения микросервисной архитектуры заключается в организации работы с большой, сложной системой, охватывающей обширную область. Но спешим вас утешить: это не уникальная проблема. В книге мы представим и используем набор практик и шаблонов, разработанных для микросервисных архитектур. Мы также покажем и реализуем инструменты, помогающие внедрять эти практики и делающие работу по созданию системы микросервисов проще, безопаснее, дешевле и быстрее.

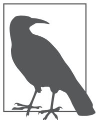
Обучение на практике

На данный момент мы установили, что микросервисы могут помочь быстрее доставлять программное обеспечение без ущерба для безопасности. Но мы также определили, что путь к хорошей микросервисной архитектуре труден и сопряжен с необходимостью принятия сложных решений. Многие из успешных разработчиков микросервисов, с которыми мы беседовали, создавали свои системы путем постоянных повторений и улучшений. Прежде чем понять, как создать работающую систему, они часто создавали неудачные архитектуры.

Будь у вас неограниченное время, вы могли бы создать отличную микросервисную архитектуру исключительно путем экспериментов, используя разные организационные модели, пробуя все методологии и создавая микросервисы различных размеров. Пока есть возможность оценивать результаты, можно продолжать совершенствовать систему. Проведя достаточное количество экспериментов, вы создадите систему, удовлетворяющую именно вашим требованиям, а также получите большой опыт создания микросервисных систем.

Однако, скорее всего, у вас ограниченный запас времени. Как же вы будете накапливать опыт, необходимый для создания более совершенных микросервисов?

Чтобы помочь решить эту проблему, мы разработали образцовую модель микросервисов. В процессе ее создания мы принимали решения о структуре команды, процессах, архитектуре, инфраструктуре, инструментах и технологиях. Мы постарались охватить широкий спектр тематических областей. Наши решения базируются на опыте создания систем микросервисов для крупных организаций. Если вы последуете нашим рекомендациям, то к концу этой книги создадите простую систему микросервисов в облачной архитектуре.



В работе мы использовали вымышленную, значительно упрощенную систему бронирования авиабилетов. Наша простая система бронирования авиабилетов будет предоставлять две функции: предоставление информации о рейсах, доступной только для чтения, и бронирование мест.

Наша цель — помочь вам как можно быстрее создать свою первую систему микросервисов. По опыту, процесс создания реальной системы — лучший способ получить истинное представление о проделанной работе и ключевых решениях. Мы не ждем, что вы согласитесь со всеми нашими решениями. На самом деле подвергать сомнению решения, которые мы приняли за вас, — важная часть пути обучения! Надеемся, модель, которую мы создадим вместе, — лишь первая из многих ваших будущих систем микросервисов.

Модель приобретения навыков Дрейфуса

Начать обучение, следуя инструкциям, — проверенный и верный путь к приобретению опыта. Согласно статье FiveStage Model of Adult Skill Acquisition Стюарта и Хьюберта Дрейфусов (https://oreil.ly/vs3ao), первый этап включает в себя следование конкретным рекомендациям для получения знаний и наработки собственного опыта.

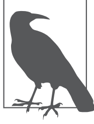
Модель микросервисов «От архитектуры до релиза»

Область применения микросервисных архитектур довольно широка. К сожалению, мы не можем рассмотреть все в одной книге, но постарались охватить тематические области, которые наиболее актуальны для системы микросервисов и больше всего влияют на успех. Перечислим, что мы будем рассматривать в модели микросервисов «От архитектуры до релиза».

- *Структура команды.* Мы приступим к работе в главе 2, начав с людей, которые должны участвовать в создании системы микросервисов. Раскроем проблемы эффективного построения команды и назовем главные факторы, влияющие на координацию микросервисов. Кроме того, представим коман-

ды, которые будем использовать в наших примерах, и инструмент Team Topologies («Топологии команд»), который поможет их создать.

- *Структура микросервисов*. После создания команд мы представим в главе 3 SEED(S)-процесс. Этот процесс проектирования позволяет создавать микросервисы, которые удовлетворяют потребности пользователей и потребителей и отличаются удобными интерфейсами и поведением. Затем, в главе 4, мы рассмотрим проблему определения правильных границ для наших микросервисов. Мы также представим некоторые важные концепции предметно-ориентированного проектирования и будем использовать процесс, называемый Event Storming, для выбора «правильного размера» наших сервисов.
- *Структура данных*. Данные — один из самых сложных аспектов проектирования микросервисов. В главе 5 мы посмотрим, какие свойства данных необходимо учитывать в системе микросервисов, представим концепцию независимости данных и заложим основу архитектуры данных для нашего проекта.
- *Облачная платформа*. Наша реализация микросервисов будет построена на облачной инфраструктуре. В главе 6 мы представим и реализуем принципы неизменяемой инфраструктуры и инфраструктуры как кода (infrastructure as code, IaC) в качестве основы для инфраструктуры микросервисов. Кроме того, мы представим облачную платформу AWS и создадим конвейер CI/CD на основе GitHub Actions. Затем, в главе 7, с помощью этого конвейера мы спроектируем и реализуем инфраструктуру микросервисов на базе AWS — она будет включать сеть, кластер Kubernetes и средство развертывания GitOps.
- *Разработка микросервисов*. Настроив инфраструктуру, мы погрузимся в задачи по разработке микросервисов. В главе 8 мы сначала опишем принципы и инструменты, необходимые для достижения успеха. Затем в главе 9 реализуем два независимых разнородных микросервиса для нашего приложения.
- *Релиз и внесение изменений*. В главе 10 мы соберем все вместе и развернем один из созданных нами микросервисов на облачной платформе. Для этого воспользуемся набором технологий, включая DockerHub, Kubernetes, Helm и Argo CD. Наконец, после выпуска рассмотрим систему в целом в главе 11.



Разработанная нами модель основана на пяти руководящих принципах, входящих в манифест «Приложение двенадцати факторов» (https://12factor.net). Если вам интересно, можете прочитать о руководящих принципах нашей модели в репозитории книги на GitHub (https://oreil.ly/MicroservicesUpand-Running).

Надеемся, этот краткий обзор помог вам получить представление о масштабах нашей модели и примере приложения. К концу книги мы реализуем полноценную систему. Но, чтобы добраться туда, нам придется принять много решений. Итак, в первую очередь нам понадобится определить способ отслеживания действительно важных решений.

Решения, решения...

В разработке ПО решения имеют большое значение. Профессиональным инженерам-программистам и архитекторам много платят за решения, которые они принимают, и за задачи, которые они решают. Качество ПО и бизнес-результаты, которые обеспечивают эти люди, зависят от качества этих решений.

Но решения не всегда легко принимать. К тому же они не всегда верны. Мы принимаем решения, наилучшие при имеющемся объеме информации, уровне опыта и таланта. Когда любая из этих переменных меняется, наши решения тоже должны меняться. Одни решения верны в определенное время, но устаревают, когда меняются технологии, люди или ситуации. Другие не были хорошими с самого начала. В любом случае нам нужен способ фиксировать важные решения, чтобы со временем мы могли их переоценить и улучшить.

Чтобы удовлетворить эту потребность, мы воспользуемся инструментом под названием *реестр архитектурных решений* (architecture decision record, ADR). Мы точно не знаем, кто изобрел термин ADR и когда он был впервые использован, но идея документирования проектных решений существует давно. Проблема заключается в том, что большинство людей не тратят на это время. По нашему опыту, ADR — чрезвычайно полезный инструмент, позволяющий прояснить принимаемые решения.

Каждая запись в реестре решений должна включать четыре важных компонента.

- *Контекст*. В чем заключается проблема, которую мы пытаемся решить? Какие ограничения имеются? Запись о принятом решении должна давать краткие ответы на эти вопросы. Благодаря этому мы сможем понять обоснование решения и почему его, возможно, потребуется обновить.
- *Альтернативы*. Решение не является таковым, если нет выбора, который нужно сделать. Хорошая запись о решении должна описывать возможные альтернативы. Это поможет нам лучше понять контекст и «пространство выбора» в момент принятия решения.
- *Выбор*. В основе решения лежит выбор. Каждая запись решения должна документировать сделанный выбор.

- *Влияние*. Решения имеют последствия, и записи решений должны документировать самые важные из них. Какие достоинства и недостатки имеет решение? Как выбор решения повлияет на рабочие подходы или на другие решения?

Вы можете оформлять записи в реестре решений так, как вам нравится: в виде текстовых файлов, использовать инструмент управления проектами или хранить их в электронной таблице. Содержание важнее, чем формат и инструментарий. Главное, чтобы каждая запись содержала четыре компонента, перечисленные выше.

В нашем демонстрационном проекте мы будем использовать формат, называемый *упрощенной записью архитектурного решения* (lightweight architectural decision record, LADR). Формат LADR, предложенный Майклом Нейгардом (https://oreil.ly/_mVoC), позволяет кратко описать принятое решение. Познакомимся с LADR поближе, создав запись вместе.

Если вы решите работать с другим форматом, отличным от LADR, то воспользуйтесь большим перечнем форматов ADR и шаблонов, которые предлагает Джоэл Паркер Хендерсон (https://oreil.ly/T3Tc-).

Создание упрощенной записи архитектурного решения

Первым ключевым решением, которое мы опишем, будет выполнение записей решений. Проще говоря, создадим ADR, в которой укажем, что намерены фиксировать принимаемые решения. Как упоминалось ранее, мы будем использовать формат LADR. Главное его преимущество — простота. Мы будем сохранять описание решений в простых текстовых файлах, а поскольку это текстовые файлы, то сможем управлять записями наших решений так же, как управляем исходным кодом.

Записи LADR оформляются на языке разметки Markdown (https://oreil.ly/oRyx0), который обеспечивает элегантный и простой способ написания документации. Самое замечательное в Markdown, что он не затрудняет чтение текста и большинство популярных инструментов знают, как его визуализировать. Например, Confluence, GitLab, GitHub и SharePoint могут представлять текст в формате Markdown и в виде удобочитаемого документа.

Чтобы создать первый документ LADR на основе Markdown, откройте текстовый редактор и создайте новый документ. Первое, что мы сделаем, — определим структуру.

Добавьте следующий текст в свой файл LADR:

```
# OPM1: Использовать ADRS для документирования решений

## Статус
Принято

## Контекст

## Решение

## Следствия
```

Это ключевые разделы записи о принятом решении. Символы # в начале строк — это маркеры Markdown, подсказывающие анализатору, что эти строки представляют собой заголовки. Обратите внимание: мы присвоили этому решению название, кратко описывающее его. Кроме того, название решения начинается со странной аббревиатуры OPM1. Это всего лишь краткий код формы, который поможет нам обозначить и понять, к какой части системы относится решение. В данном случае OPM1 указывает, что это первое наше решение и оно связано с операционной моделью (OPerating Model).

Подзаголовок `Статус` дает понять, на какой стадии жизненного цикла это решение находится. Например, если вы разрабатываете новое решение, по которому необходимо получить согласие, то первоначально ему можно присвоить статус `Предложено`. Или, если вы рассматриваете возможность изменения существующего решения, можете изменить его статус на `На рассмотрении`. В нашем случае мы уже приняли решение, поэтому установили статус `Принято`.

В разделе `Контекст` описывается проблема, ограничения и предпосылки для принятого решения. В данном случае мы постарались отразить необходимость регистрации важных решений и объяснить, почему это важно. Добавьте следующий текст (или его аналог) в раздел `Контекст` вашей записи:

```
## Контекст
Архитектура микросервисов сложна, и потребуется принять множество решений. Нам
понадобится способ отслеживать важные принимаемые решения, чтобы можно было
пересмотреть и переоценить их в будущем. Мы предпочли бы использовать простой
текстовый формат описания решений, чтобы не пришлось устанавливать какие-либо
новые программные средства.
```

Описав контекст, можно переходить к описанию фактического принятого нами решения, перечислению некоторых из рассмотренных альтернатив, а также обоснованию выбора в пользу LADR. Добавьте следующий текст в раздел `Decision`, чтобы задокументировать этот факт:

```
## Решение
Мы решили использовать формат упрощенной записи архитектурных решений
Майкла Найгарда — LADR. Этот текстовый формат достаточно легковесный,
чтобы удовлетворить наши запросы. Мы будем хранить каждую LADR в отдельном
текстовом файле и управлять файлами как исходным кодом. Мы также
рассмотрели следующие альтернативные решения:
* инструменты управления проектами (отвергнуто, поскольку мы бы не хотели
устанавливать дополнительные инструменты);
* неформальное ведение записей или сарафанное радио (ненадежно).
```

Осталось задокументировать следствия. В нашем случае одним из ключевых следствий является необходимость тратить время на документирование решений и управление записями. Зафиксируем это следующим образом:

```
## Следствия
* Нам потребуется создавать записи решений для ключевых решений.
* Понадобится инструмент для управления файлами записей решений
как исходным кодом.
```

Вот и все, что нужно для оформления LADR. Это невероятно полезный способ документирования мыслей, к тому же он имеет дополнительное преимущество, заставляя принимать взвешенные и обдуманные решения. По мере создания нашего примера приложения бронирования авиабилетов мы будем вести реестр принятых нами ключевых решений. Однако для экономии времени вместо создания полноценных записей о принятии решений мы будем отмечать их вот такими примечаниями.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ ADR ДЛЯ ОТСЛЕЖИВАНИЯ РЕШЕНИЙ

Мы будем создавать записи в реестре ADR для фиксации ключевых решений, принятых при проектировании и создании нашей системы.

Полные версии записей о принятии решений вы найдете в репозитории GitHub для этой книги (https://github.com/implementing-microservices/ADRs).

Резюме

В этой главе мы познакомили вас с некоторыми основополагающими концепциями. Дали общее определение системы микросервисов как набора трех ключевых характеристик. Обозначили сокращение затрат на координацию как главное преимущество микросервисов. Кроме того, мы обсудили, как

сложность и аналитический паралич создают проблемы для пользователей микросервисов.

Чтобы помочь решить эти проблемы, мы разработали модель микросервисов «От архитектуры до релиза», которая ускорит процесс обучения разработчиков. Рассмотрели аспекты модели и темы, которые обсудим. Наконец, представили концепцию реестра архитектурных решений (ADR), который планируем использовать далее в книге.

Теперь, закончив обзорную часть, перейдем к созданию системы. В главе 2 мы начнем знакомиться с особенностями организации микросервисов и уделим особое внимание координации деятельности команд.

ГЛАВА 2

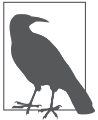
Разработка операционной модели микросервисов

В этой книге мы создадим приложение на основе микросервисов, спроектировав и реализовав микросервисы, а также инфраструктуру и инструменты, необходимые для их поддержки. Однако для успешного внедрения микросервисов недостаточно просто написать код и развернуть его. Чтобы добиться успеха и заставить всю систему работать, необходимы люди, обладающие нужными знаниями и применяющие правильные методы работы и принципы. Вот почему мы хотим начать повествование с разработки общей операционной модели для нашего приложения.

Операционная модель — это люди, процессы и инструменты, лежащие в основе вашей системы. Эта модель определяет все процессы принятия решений и работу, выполняемую при создании программного обеспечения. Например, операционная модель может определять обязанности команд. Кроме того, она регулирует управление процессом принятия решений и работой.

Вы можете представлять себе операционную модель как «операционную систему» для вашего решения. Вся работа по созданию микросервисов выполняется поверх определяемых вами командных структур, процессов и границ. На практике операционные модели могут иметь большой охват и быть очень подробными. Но для целей нашей дискуссии мы уменьшим масштаб и сосредоточимся на наиболее важных частях системы микросервисов — на том, как создаются команды и как они работают во взаимодействии.

Это и есть тема, которую мы рассмотрим в текущей главе: взаимосвязь между командами и реализациями микросервисов. Далее мы представим инструмент Team Topologies — и к концу главы получим организационную структуру команды, которую сможем использовать как основу для дальнейшего обсуждения.



На самом деле вам необязательно собирать людей и команды, которые мы определим, чтобы следовать нашей модели создания микросервисов «От архитектуры до релиза».

Начнем с того, что обсудим, почему команды и их структура имеют первоочередную важность.

Почему команды и люди важны

Модель, которую мы разбираем в книге, в основном затрагивает решения в области технологий и инструментов. Но технология сама по себе не дает той ценности, которая требуется от системы микросервисов. Технологии очень важны. Правильный их выбор облегчает выполнение задач, которые в ином случае были бы непомерно трудными. В лучшем случае технологии открывают двери и раскрывают новые возможности. Однако сами по себе технологии бесполезны.

У вас могут быть лучшие в мире инструменты и платформы, но вы потерпите неудачу, если у вас нет правильной культуры и организации, в которых вы можете их применить. Цель, которую мы пытаемся достичь в нашей модели, состоит в том, чтобы передать хорошие технологии в руки независимых, хорошо функционирующих команд. Итак, нам стоит начать с рассмотрения типов команд и структуры, которая лучше всего подойдет для разрабатываемой нами модели.

В системе микросервисов важны культура и структура команды. В ходе исследований для книги и исходя из собственного опыта внедрения мы усвоили важную истину: люди и процесс — главные факторы успеха. Реализовать микросервисную архитектуру имеет смысл, когда она дает вам свободу легко и быстро вносить изменения. Однако на практике изменения — это побочный продукт способности вашей организации принимать решения. Не имея возможности быстро принимать эффективные решения, вам будет трудно получить выгоду от своих микросервисов. Это все равно что создать гоночный автомобиль с очень плохим двигателем. Независимо от того, насколько хорош сам автомобиль, он никогда не будет ехать так, как должен.

Идея о важности структуры и культуры важна и не нова. Мел Конвей красноречиво описал влияние структуры команды на системный проект в своей известной статье *How Do Committees Invent?* (https://oreil.ly/oRyx0). Всем знаком закон Конвея, который гласит:

> «Любая организация проектирует систему (определяемую в широком смысле), которая копирует структуру коммуникаций в этой организации».
>
> *Приписывается Фреду Бруксу*

Конвей говорит, что результаты деятельности организации отражают то, как общаются ее сотрудники и команды. Например, рассмотрим команду, отвечающую за микросервисы, которая должна консультироваться с централизованной командой экспертов по базам данных всякий раз, когда необходимо изменить модель данных. Скорее всего, модель и ее реализация тоже будут централизованы в создаваемой системе. В итоге она соответствует модели организации и координации.

Вывод из всего этого: люди в системе микросервисов очень важны. То, как они принимают решения, выполняют свою работу и общаются друг с другом, очень сильно влияет на создаваемую систему. В целом есть три человеческих фактора, оказывающих наибольшее влияние на систему микросервисов: размер команды, навыки работы в ней и координация между командами. Рассмотрим их подробнее.

Размер команды

«Микро» в микросервисах подразумевает, что размер имеет значение и чем он меньше, тем лучше. Честно говоря, это чрезмерное упрощение. Но правда остается правдой: создание небольших развертываемых сервисов — важная часть успеха внедрения микросервисов. К тому же, как оказывается, размер команд, создающих эти сервисы, тоже имеет большое значение.

Если в команде слишком много людей, то им придется тратить больше времени на общение друг с другом. Эта внутренняя координация в итоге замедлит работу команды, что приведет к замедлению внедрения изменений. Если людей слишком мало, то не хватит рук, чтобы выполнить работу. Выбор «правильного размера» команд — важная часть проектирования системы. Хотя не существует универсального размера, подходящего всем и во всех ситуациях, исследования размеров команд превратились в общепринятую практику.

Билл Гор, соучредитель компании W. L. Gore по производству материалов Gore-Tex, ограничил размер команд в компании в целях сохранения их эффективности. Чтобы добиться этого, он установил обязательное ограничение по размеру: у каждого члена команды должны быть личные отношения (https://oreil.ly/wduQE) друг с другом. Когда команда достигает такого масштаба, что ее участники не знают друг друга, подразделение считается слишком большим.

Антрополог Роберт Данбар в своих исследованиях социального поведения шимпанзе заметил, что размеры групп шимпанзе коррелируют с размером их мозга. Экстраполируя эти результаты на свое понимание человеческого мозга, исследователь определил, каких размеров должны быть команды людей. Число

Данбара (https://oreil.ly/-DbyT) определяет, что, исходя из размера нашего мозга, мы можем комфортно поддерживать только 150 стабильных отношений.

Данбар также определил, что люди могут поддерживать тесные отношения примерно с пятью членами семьи, а всего с 15 близкими людьми.

Возможно, самым известным руководством к формированию команд стало «правило двух пицц» (https://oreil.ly/ccT85), сформулированное генеральным директором Amazon Джеффом Безосом. В нем говорится, что команда должна быть настолько маленькой, чтобы ее можно было накормить двумя пиццами. Хотя конкретные детали о размере пиццы и аппетите участников неясны, такая команда, вероятно, состоит из 5–15 человек. Именно такой диапазон описывает Данбар, и именно в такой команде есть хорошие шансы сохранить эвристику личных отношений, которую описывает Гор.

Все эти истории указывают на ограничение по размеру, основанное на способности людей эффективно общаться. Наш опыт и исследования согласуются с этой интуитивной концепцией. Чтобы поддерживать высокую скорость изменений, нужно ограничить размер команд в системе. В нашей модели микросервисов мы собираемся установить размер команд где-то от пяти до восьми человек.

КЛЮЧЕВОЕ РЕШЕНИЕ: РАЗМЕР КОМАНД ДОЛЖЕН БЫТЬ ОГРАНИЧЕН

В каждой команде, работающей в нашей системе, должно быть не более восьми человек.

Ограничение численности команды поможет сократить необходимые внутренние взаимодействия. Но это ограничение имеет побочный эффект. Чем меньше размер одной команды, тем больше должно быть команд. Поэтому нужно быть осторожными в проектировании остальной части системы. Небольшие команды будут бесполезны, если им придется тратить все свое время на координацию друг с другом. Чтобы избежать этого, нужно обеспечить максимально безопасную независимую и автономную работу.

Еще один побочный эффект ограничения размеров команд — такой подход ограничивает количество специалистов, которых мы можем подключить к работе. Располагая меньшим количеством людей в команде, необходимо убедиться, что сотрудников с определенными навыками достаточно для получения качественного результата. Вот почему нам нужно будет рассмотреть наполнение команд с точки зрения квалификации.

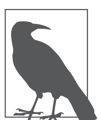
Мастерство команды

Команда может быть хороша ровно настолько, насколько хороши ее участники. Если мы хотим получить высокоэффективную команду, то нужно уделить особое внимание ее составу. Например, какие роли и специализации понадобятся командам? Насколько способными и опытными должны быть отдельные участники? Какое сочетание навыков и опыта можно считать правильным?

Правда в том, что на эти вопросы трудно дать универсальные ответы. Это потому, что люди и культура труда часто являются самой уникальной чертой места, где вы работаете. Например, некоторые компании тратят много денег, чтобы нанять 1 % лучших специалистов в области технологий со всего мира. Другие могут нанимать местных талантов, предполагая их профессиональный рост и обучение на рабочем месте у небольшого количества экспертов. Хорошая структура команды в этих двух случаях, вероятно, будет выглядеть совершенно по-разному.

Но эта книга посвящена созданию микросервисов, поэтому не будем углубляться в организационную и культурную структуру. Но мы можем принять общий принцип, который поможет разработчикам микросервисов повсеместно. Это принцип кросс-функциональной команды.

В кросс-функциональной команде люди с разными знаниями (или функциями) работают вместе для достижения одной цели. Эти знания могут охватывать как технологические, так и бизнес-области. Например, кросс-функциональная команда может состоять из UX-дизайнеров, разработчиков приложений, владельцев продуктов и бизнес-аналитиков.



Кросс-функциональные команды существуют давно, по крайней мере с 1950-х годов, и впервые появились в страховой компании Northwestern Mutual.

Строить команду таким образом — значит иметь большое преимущество. Оно заключается в возможности быстрее принимать правильные решения. Мы уже установили верхний предел размера команды, ограничив численность восемью участниками. Команда «правильного размера» с нужными людьми может уверенно продвигаться в работе с высокой скоростью.

Но кто эти *нужные люди*? Когда мы обсуждали размер команды, то опирались на свои наблюдения, опыт и научные исследования. Но в отношении состава команд гораздо сложнее найти повторяющиеся примеры. В частности, крупный поставщик облачных услуг, работая над микросервисами, может привлечь

4–5 многопрофильных специалистов и одного тестировщика. И наоборот, консалтинговые компании могут использовать в каждой команде множество инженеров-специалистов, владельцев продуктов, менеджеров проектов и тестировщиков. Точный состав специалистов будут определять умения, опыт и производственная культура вашей организации.

Итак, вместо того, чтобы указывать вам, какие роли назначать в командах, мы примем два общих решения для нашей модели. Во-первых, команды должны быть многофункциональными. Наш опыт показывает, что наибольшего успеха в создании микросервисных архитектур добиваются команды, способные самостоятельно принимать правильные решения. Кросс-функциональные команды готовы к этому. Во-вторых, команды должны состоять из тех, кто непосредственно влияет на результат. Поэтому мы подберем людей, которые, как уже известно, могут принести пользу команде. Нам не нужны наблюдатели или люди, имеющие лишь косвенное отношение к работе и принимаемым решениям.

КЛЮЧЕВОЕ РЕШЕНИЕ: ДОЛЖНЫ БЫТЬ ОПРЕДЕЛЕНЫ ПРИНЦИПЫ ЧЛЕНСТВА В КОМАНДЕ

Команды должны быть кросс-функциональными и состоять только из тех, кто может повысить ценность результатов, сервисов или продукта команды.

Определив правильный размер команды и то, какие специалисты нужны, мы сможем создать эффективные команды, способные добиваться поставленных целей. По мере роста количества команд нам также необходимо будет рассмотреть вопрос о координации своих действий друг с другом. Это последнее свойство команды, которое нужно рассмотреть.

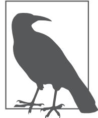
Взаимодействие между командами

Именно общение между командами, а не внутри них может застопорить систему микросервисов. Мы осветили проблему затрат на координацию в подразделе «Проблема затрат на координацию» главы 1. Если нам удастся сократить объем согласований между командами, то они смогут быстрее вносить изменения.

Было бы неплохо, если бы наши команды могли действовать полностью автономно и независимо. Будь у команд свобода принятия решений по проектированию, разработке, тестированию и развертыванию, не было бы никаких «организационных трений», которые замедляли бы работу. Но, по нашему опыту, это непрактичный метод работы.

Все потому, что координация и сотрудничество важны для успеха организации. И мы хотим, чтобы команды действовали независимо, но нам также важно, чтобы они создавали сервисы, важные для клиентов, пользователей и организации. Это значит, что коммуникация необходима для установления общих целей, передачи запросов на изменения, предоставления обратной связи и решения проблем.

Вдобавок ко всему, когда команды работают полностью независимо, у них меньше возможностей для обмена информацией. Команды, работающие независимо, могут выбирать правильные инструменты и создавать высокоэффективные системы. Но эта эффективность локализована в команде. Иногда это означает потерю эффективности на *системном уровне*. Например, если все команды разрабатывают и создают собственные облачные сетевые архитектуры, мы теряем возможность выполнить эту работу один раз и поделиться ею.



Можно построить организацию, которая эффективно обеспечивает независимость и автономию команды за счет самоорганизации. Например, основоположник микросервисов Фред Джордж описал метод, который он называет Programmer Anarchy (https://oreil.ly/C1N0f). В этом случае технические специалисты обладают полной автономией (и ответственностью) в формировании команд, выборе работы и разработке собственных решений. Но, по нашему опыту, большинству корпоративных организаций трудно постоянно придерживаться этого правила.

Если зайти слишком далеко в обеспечении независимости и автономии команды, то это повлечет неэффективность на системном уровне и несоответствие целям организации. Если на координацию будет тратиться слишком много времени, мы рискуем затормозить всю систему и потерять преимущества легко изменяемых микросервисов. Наша задача — найти правильный баланс между независимостью и координацией усилий. Это требует некоторых экспериментов и постоянной подстройки структуры вашей команды.

Самое главное, оптимизация координации команды требует активных усилий по проектированию. Одна из ошибок, которые, как мы видели, совершают практикующие специалисты, — сосредоточение исключительно на технической архитектуре. Когда это происходит, структура команды формируется вокруг созданной технологии. Только тогда проблемы с координационной моделью становятся очевидными. К этому моменту часто вносить изменения становится слишком дорого или слишком сложно.

Чтобы избежать этой проблемы, мы рассмотрим координацию команды и ее структуру в качестве первого шага нашего процесса проектирования системы.

Некоторые называют это «обратным маневром Конвея», поскольку структура коммуникаций, которую мы разрабатываем, в итоге будет определять создаваемую систему. Мы обнаружили, что, начав с акцента на структуре команды и координации, действительно можно добиться успеха в разработке микросервисов. На самом деле этот момент настолько важен, что мы задокументируем его как решение.

КЛЮЧЕВОЕ РЕШЕНИЕ: КОГДА ОПРЕДЕЛЯТЬ СТРУКТУРУ КОМАНД И МОДЕЛИ КООРДИНАЦИИ

Обсуждать структуру команды и способы координации нужно до проектирования архитектуры системы или микросервисов. Модели команды и способы взаимодействия должны постоянно обновляться и совершенствоваться в течение всего срока службы системы.

Мы поговорим об этом в оставшейся части главы. Но сначала представим полезный инструмент для проектирования моделей команд микросервисов — Team Topologies.

Введение в Team Topologies

Поскольку мы собираемся начать нашу проектную работу, сосредоточившись на командах, нам понадобится способ каталогизации наших решений и информирования о них. Существует множество способов документирования структуры команды. Для создания модели мы используем инструмент проектирования Team Topologies («Топологии команд»).

Team Topologies (https://teamtopologies.com) — подход к проектированию, изобретенный Мэтью Скелтоном и Мануэлем Пейсом. Нам нравится использовать его, поскольку он предусматривает формальный язык для обсуждения структуры команды, позволяя делать особый акцент на том, как команды работают друг с другом.

Мы не станем использовать все аспекты подхода топологий команд в нашей проектной работе. Вместо этого мы будем опираться на три элемента: типы команд, режимы взаимодействия в команде и схематичное изображение. С помощью этих компонентов можно определить простую работающую структуру для наших команд микросервисов.

Далее мы рассмотрим различные части подхода топологий команды, начиная с типов команд, которые можем определить.

Тип команды

Одно из основных понятий топологий команд — *типы команд*. Это архетипы или категории, которые описывают природу команды с точки зрения ее взаимодействия с остальной частью организации. В топологиях команд определены четыре типа команд. Рассмотрим каждый из них.

- *Команда, ориентированная на поток (потоковая)*, владеет и управляет модулем системы. Ключевой характеристикой этой команды является постоянное предоставление чего-то, что имеет отношение к бизнес-организации. Потоковая команда — воплощение цитаты (https://oreil.ly/bIwkK) технического директора Amazon Вернера Фогеля об обязанностях команд Amazon: «Ты создал, ты и управляешь». Потоковые команды не распускаются после релиза. Вместо этого они продолжают управлять и внедрять «поток» изменений, улучшений и исправлений в результаты своего труда. Например, команды, работающие с микросервисами, обычно являются потоковыми, поскольку постоянно совершенствуют принадлежащие им сервисы.
- *Команда поддержки (поддерживающая)* обеспечивает работу других команд путем взаимодействия с консультантами. Такие команды обычно состоят из предметных специалистов, которые могут устранить пробелы в знаниях или возможностях. Но они также могут помочь отдельным командам понять общую картину организации или отрасли, в которой те работают. Например, группа поддержки архитектуры может помочь командам, отвечающим за микросервисы, разобраться в новых технических стандартах и соглашениях в организации.
- *Команда разработки сложных подсистем* работает над областью или предметом, который трудно понять. Или по крайней мере с тем, что достаточно сложно сделать из-за нехватки доступных ресурсов в организации. Некоторые проблемные области плохо масштабируются и не могут быть внедрены в каждую команду. Например, настроить программное обеспечение для криптографической безопасности можно, только имея специальные знания и опыт. Вместо того чтобы пытаться распределить эти навыки между всеми командами, большинство организаций создают команду разработки сложной подсистемы безопасности, которая по мере необходимости взаимодействует с другими командами.
- *Команда разработки платформы*, как и команды поддержки, обеспечивает поддержку остальной части организации, но с одним важным отличием: она предоставляет своим пользователям возможность самообслуживания. В то время как команды поддержки и сложных подсистем ограничены пропускной способностью, обеспечиваемой сотрудниками, работающими в команде,

команда разработки платформы вкладывается в создание вспомогательных инструментов и процессов, которые могут легко масштабироваться. Это требует больших первоначальных вложений, а также постоянного сопровождения и поддержки. Платформа становится продуктом, которым пользуются остальные команды организации. Например, команды по эксплуатации могут стать командами разработки платформы, когда предлагают командам разработчиков инструменты сборки и выпуска.

Имея представление об этих четырех типах команд, мы можем начать использовать тот тип коммуникации, к которому хотели бы подвести свои команды. Чтобы действительно объяснить нашу структуру команды, мы должны обсудить еще одну часть модели: способы взаимодействия команд. Их мы и рассмотрим далее.

Режимы взаимодействия

Наша цель при проектировании команд для сборки микросервисов — уменьшить объем координации, необходимый для выполнения работы. Типы согласно топологии помогают определить основные характеристики команды. Чтобы действительно понять, как и где можно снизить затраты на координацию, нужно четко сформулировать, как наши команды координируют свои действия друг с другом. Именно здесь и пригодятся режимы взаимодействия согласно топологии команд. В своей книге Скелтон и Пейс обсуждают три режима взаимодействия, которые описывают различные уровни координации.

- *Сотрудничество*. Этот режим взаимодействия предполагает тесное сотрудничество обеих команд. Благодаря ему у команд появляются возможности для обучения, открытий и инноваций. Но он требует высокого уровня координации от каждой команды, и его трудно масштабировать. Например, команда, отвечающая за безопасность, может сотрудничать с командой разработки микросервисов для создания более безопасной версии программного обеспечения. Сотрудничество может включать совместное проектирование, написание и тестирование кода.
- *Поддержка*. Взаимодействие в форме поддержки похоже на сотрудничество, но является однонаправленным. Вместо совместной работы над решением общей проблемы одна команда поддерживает другую, помогая ей достичь желаемого результата. Примером взаимодействия по принципу поддержки может служить ситуация, когда команда инфраструктуры помогает команде микросервисов понять, как устранять неполадки в предоставленной им сетевой архитектуре.
- *«Все как сервис»* (X-as-a-service, XaaS). Иногда сотрудничество команд приобретает характер взаимодействия между потребителем и поставщи-

ком. При таком типе взаимодействия одна команда предоставляет услугу другим командам в организации с минимальным уровнем координации. Обычно такие взаимодействия имеют место, когда команда готовит общий процесс, документ, библиотеку, API или платформу. Взаимодействия типа XaaS, как правило, хорошо масштабируются, поскольку требуют меньшей координации. Они также идеально подходят для команд разработки платформы, но другие типы команд тоже могут использовать этот режим. Например, команда поддержки архитектуры может задокументировать список рекомендуемых шаблонов программного обеспечения и предложить их всем командам, работающим над микросервисами, в модели «Шаблоны как сервис».

Возможности топологии команд гораздо шире, чем мы описали. В совокупности эта классификация типов команд и взаимодействий дает нам большую палитру терминов. С их помощью можно нарисовать картину того, как должны выглядеть наши команды микросервисов, сделав особый акцент на том, когда и в каком объеме командам потребуется координация. В следующем разделе мы будем использовать термины, позаимствованные из топологии команд, для разработки модели команды микросервисов.

Разработка топологии команды по работе с микросервисами

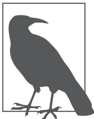
Подход топологии команд предоставляет язык для обсуждения взаимодействия команды. Особенность этого языка в том, что он создан для визуальных представлений. В данном разделе мы создадим структуру наших команд микросервисов, которая расскажет нам, какие команды необходимы и как они будут работать вместе. Закончив, мы получим диаграмму, на которой будут выделены основные моменты координации и взаимодействия команды.

Проектируя топологию команды, мы будем следовать пошаговому подходу, описанному ниже.

1. Образовать команду разработки системы.
2. Создать шаблон для будущих команд по работе с микросервисами.
3. Определить команды разработки платформы.
4. Добавить команду поддержки и команду разработки сложных подсистем.
5. Добавить ключевые группы потребителей.

По мере прохождения каждого из этапов мы будем документировать структуру нашей команды и создавать ее топологию. На каждом шаге мы определим одну

или несколько команд, создадим и заполним проектный документ команды и изобразим ключевые взаимодействия для нее. Начнем с команды разработки системы.



Не существует единой топологии команды, которая подходила бы всем. Было бы невозможно учесть размер вашей организации, людей, навыки и потребности. Топология, которую мы создали здесь, представляет собой консолидированную версию крупномасштабных корпоративных реализаций, которые, как мы видели, хорошо работают.

Создание команды разработки системы

Система микросервисов — это сложная система с множеством частей и большим количеством людей, выполняющих работу. Создаваемое программное обеспечение возникает в результате коллективного принятия решений и совместной работы всех этих людей. По нашему опыту, заставить все работать должным образом непросто. Вот почему нужно назначить группу людей, которые могут формировать видение и поведение системы. В нашей модели мы назовем эту группу командой разработки системы.

В нашей модели у этой команды есть три основные обязанности.

- *Разработать структуру команды.* Команда разработки системы — это первая команда, которую мы собираем. Мы также ожидаем, что именно она сформирует структуру команд, выполняющих работу по созданию системы. Это та работа, которую мы будем выполнять на последующих этапах создания структуры команды. По сути, мы вместе играем роль команды разработки системы.
- *Задавать стандарты, стимулы и «ограничения».* В дополнение к структурированию команд команда разработки системы должна формировать решения, которые могут принимать отдельные команды. Это гарантирует, что команды добьются результатов, соответствующих нашим системным целям. Один из способов сделать это — ввести стандарты, которые диктуют, что команды могут и чего не могут делать. Это предписывающий подход, который мы использовали для принятия многих решений в книге. На практике слишком обширную стандартизацию трудно поддерживать, и она слишком ограничивающая для здоровой системы. Хорошие разработчики будут вводить стимулы для получения желаемого поведения и «ограничения», действующие скорее как рекомендации и ориентиры, а не явные правила.
- *Постоянно совершенствовать систему.* Наконец, команда разработки системы должна постоянно совершенствовать структуры команд, стандарты,

стимулы и ограничения, которые были введены. Для этого они должны разработать способ мониторинга и оценки системы в целом, чтобы получить возможность вносить изменения и внедрять улучшения.

Обязанности команд желательно задокументировать, чтобы четко понимать, что делает каждая команда. На самом деле стоит документировать все ключевые свойства наших команд, чтобы их было легче понять и улучшить по мере развития системы. Как минимум мы должны описать тип согласно принципу топологии команд, размер команды и обязанности, которые мы определили ранее.

Начнем с выбора типа топологии команды. Установив структуру и стандарты команды, мы ожидаем, что команда разработки системы сосредоточится на оказании помощи другим командам в создании микросервисов и вспомогательных компонентов. Мы ожидаем, что бо́льшая часть их работы будет основана на консультациях, содействии командам развертывания и оказании им помощи в навигации по системе. Несмотря на то что команда разработки системы создает систему, работа, которую выполняют ее члены, характерна для команды поддержки.

Нам также требуется, чтобы команда разработки системы была небольшой. Она должна включать лишь нескольких старших разработчиков, архитекторов и системных проектировщиков, которые могут быстро принимать совместные решения для системы в целом. С этой целью мы ограничим размер команды 3–5 людьми — даже меньше размера типовой команды, определенного выше.

Зафиксируем эти решения и свойства команды, создав упрощенный проектный документ для команды разработки системы. Используя текстовый редактор или редактор документов, создайте файл `system-design-team.md` и заполните его следующим содержимым:

```
# Команда создания системы

## Тип команды
Поддержка

## Размер команды
3-5 человек

## Отвечает за
* разработку структуры команды
* установку стандартов и "ограничений"
* постоянное совершенствование системы
```

Преимущество использования текстовых файлов для документирования состоит в том, что их можно рассматривать как код, хранить в репозитории и обновлять всякий раз, когда нам потребуется внести изменения. В качестве

альтернативы можно использовать хранилище документов wiki или нечто иное, лучше всего подходящее для вашей компании. Мы оставляем за вами право решать, как управлять проектными файлами команд. Вы можете найти все примеры проектов нашей команды в репозитории GitHub (https://oreil.ly/Microservices_UpandRunning_team-designs).

На данном этапе мы обычно составляем визуальную схему команды и намечаем ее взаимодействия с другими командами в системе. Это основа работы по созданию структуры команды. Она позволяет визуально представить, как команды будут работать вместе. Например, мы можем ожидать, что команда разработки системы будет использовать упрощенную модель взаимодействий с командами микросервисов. Но, поскольку это первая команда, которую мы определили, нам не с чем взаимодействовать. Так что мы оставим составление диаграмм на потом.

Создав документ команды разработки системы, мы можем перейти к документированию и построению диаграмм команд по созданию микросервисов.

Подготовка шаблона команды по созданию микросервисов

В модели «От архитектуры до релиза» каждый микросервис закреплен за конкретной командой. Эти команды принимают решения, проектируют, развертывают и сопровождают микросервисы. На практике одна команда может управлять несколькими микросервисами. Это нормально и позволяет избежать ненужного роста количества команд. Но есть важное ограничение — ответственность за микросервис не распределяется между несколькими командами. Владение микросервисом ограничено подотчетной и ответственной командой.

> **КЛЮЧЕВОЕ РЕШЕНИЕ: ВЛАДЕНИЕ МИКРОСЕРВИСОМ**
>
> Каждый микросервис будет принадлежать единственной команде, которая будет его проектировать, создавать и запускать. Эта команда отвечает за микросервис в течение всего срока его службы.

По мере развития системы у вас появится множество микросервисов. Также вполне вероятно, что в конечном счете вы получите множество команд по работе с микросервисами. Мы ожидаем, что в микросервисной системе будет работать несколько команд, поэтому не будем проектировать каждую из них по отдельности, а просто определим шаблон команды микросервисов, применимый к любым вновь создаваемым командам. Этот шаблон можно рассматривать как форму для печенья, которую можно использовать, чтобы «штамповать» новые команды микросервисов позже, когда они понадобятся. Или, если у вас есть опыт

программирования, представляйте этот шаблон как определение «класса», на основе которого позже будут создаваться «экземпляры».

Для начала сделаем тот же шаг, что и при создании команды разработки системы, — определим некоторые важные свойства команды. Как и раньше, задокументируем тип команды, ее размер и обязанности. Как упоминалось ранее, ожидается, что команды микросервисов будут владеть одним или несколькими микросервисами независимо. Это владение предполагает запуск сервиса и выпуск непрерывного потока улучшений, исправлений и изменений по мере необходимости.

С учетом этой характеристики имеет смысл классифицировать команду микросервисов как потоковую. Мы также будем придерживаться решения о размере команды, принятого ранее в этой главе, и сохраним численность команды от пяти до восьми человек. Задокументируем все эти свойства, как делали раньше. Создайте файл `microservice-team-template.md` и заполните его следующим содержимым:

```
# Шаблон команды разработки микросервисов
## Тип команды
Потоковая
## Размер команды
5-8 человек
## Отвечает за
* проектирование и разработку микросервиса (-ов)
* тестирование, отладку и развертывание микросервиса (-ов)
* устранение неполадок
```

Описав шаблон, можно приступать к построению диаграммы модели взаимодействий команды. Для этого откройте инструмент рисования или построения диаграмм и нарисуйте горизонтальный прямоугольник, как показано на рис. 2.1. Команды этого типа мы будем обозначать желтым цветом, а вообще команды каждого типа должны обозначаться своим цветом (в книге все будет показано оттенками серого. — *Примеч. ред.*).

Команда по работе с микросервисами

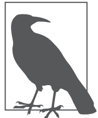
Рис. 2.1. Потоковая команда по работе с микросервисами

Если у вас нет любимого инструмента для построения диаграмм, то мы можем посоветовать diagrams.net (https://www.diagrams.net) и Lucidchart (https://www.lucidchart.com) — хорошие и бесплатные браузерные варианты. Конечно, вы также можете рисовать диаграммы по старинке, с помощью ручки и листа бумаги!

Выше мы определили нашу команду разработки системы. Теперь, когда у нас есть диаграмма для команды по работе с микросервисами, можно добавить в картину системную команду. Нарисуйте команду разработки системы, используя вертикальный прямоугольник, как показано на рис. 2.2.

Рис. 2.2. Добавление команды разработки системы

Используйте уникальный цвет (мы использовали фиолетовый) для команды разработки системы, чтобы показать, что это команда поддержки. Мы разместили ее вертикально и слева от команды микросервисов, чтобы показать взаимодействие между двумя командами. В этом случае предполагается, что команда разработки системы будет помогать командам микросервисов. Для простоты мы не будем моделировать конкретные детали взаимодействий. Сейчас достаточно подчеркнуть, что команды должны будут взаимодействовать.



Наш выбор цвета для типов команд в этой главе основан на иллюстрациях, показанных на сайте Team Topologies (https://oreil.ly/dgEUz).

На практике по мере развития системы потребуется заменить это обобщенное поле «Команда по работе с микросервисами» фактическими названиями ваших команд и сервисов, над которыми они работают. Со временем вам также может потребоваться фиксировать взаимодействия, которые должны происходить между командами микросервисов. Например, если одному микросервису необходимо вызвать другой, то, скорее всего, потребуется дополнительная работа по координации, которую стоит зафиксировать.

Конкретный цвет указывает, что команда по работе с микросервисами — потоковая. Мы будем обновлять эту диаграмму по мере прохождения этапов создания структуры команды, поэтому в дальнейшем держите инструмент для рисования под рукой. Вы также можете сохранить диаграмму, чтобы не потерять проделанную работу.

Теперь, смоделировав первые две команды, взглянем на команду разработки облачной платформы.

Команды разработки платформы

Команды разработки платформы — важная часть микросервисной системы. Бóльшая часть работы с микросервисами выполняется независимыми потоковыми командами. Однако без поддержки им придется самостоятельно решать множество проблем с разработкой, тестированием и внедрением. Наша вспомогательная команда разработки системы может помочь им в принятии некоторых решений, но командам микросервисов все равно придется иметь дело со сложностями всего технологического стека и архитектуры.

Именно здесь могут помочь команды, относящиеся к платформенному типу. В системе микросервисов есть много общих компонентов. Команда разработки платформы может сделать эти общие компоненты доступными для микросервисов «как услуги». Модель обслуживания улучшает масштабируемость компонентов платформы, уменьшая проблемы координации, которые обычно возникают при централизации общих компонентов.

В нашей модели мы решили создать команду разработки облачной платформы, которая предоставляет сеть, приложения и инфраструктуру развертывания для остальной части организации как услугу. Мы подробно рассмотрим, как выглядит это предложение, в главе 7, когда углубимся в проектирование инфраструктуры. Ключевым моментом на данном этапе является возможность создания командами новых окружений по требованию, используя сервисы инфраструктуры, предоставляемые командой разработки платформы.

Поняв эти детали, можно задокументировать команду разработки облачной платформы в файле `cloud-platform-team.md` со следующими свойствами команды:

```
# Команда облачной платформы

## Тип команды
Команда разработки платформы

## Размер команды
5-8 человек
```

```
## Отвечает за
* проектирование и разработку сетевой инфраструктуры
* проектирование и разработку инфраструктуры приложений
* предоставление инструментов для создания новой среды
* обновление сетевой и прикладной инфраструктуры по мере необходимости
```

Обратите внимание, что одна из обязанностей команды разработки облачных платформ — обновление предлагаемой инфраструктуры. Это ключевая часть ответственности данной команды. К пользователям платформы следует относиться так, как если бы они были заказчиками. В этих взаимоотношениях предложения команды разработки платформы должны постоянно совершенствоваться, чтобы соответствовать требованиям и ожиданиям их клиентов.

Как и прежде, добавим команду разработки облачной платформы в схему топологии команды. Для этого нарисуйте горизонтальный прямоугольник (опять же используя уникальный цвет; мы выбрали голубой) под командами микросервисов и подключите его к команде разработки системы, как показано на рис. 2.3.

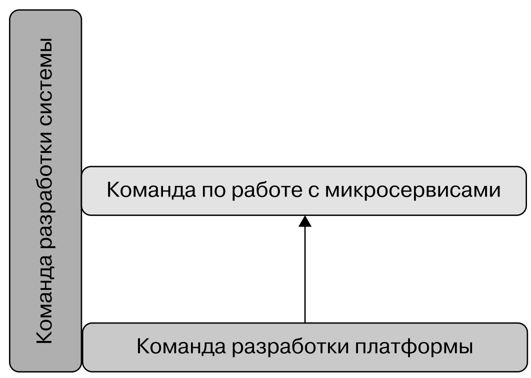


Рис. 2.3. Команда разработки облачной платформы, предлагающая сервис

Обратите внимание, что мы также нарисовали маленькую черную стрелку между командами платформы и микросервисов, чтобы показать, что команда разработки платформы использует модель XaaS для своего взаимодействия с командой микросервисов. На нашей диаграмме также показано, что команда разработки системы будет обеспечивать работу команды разработки платформы. Это гарантирует, что платформа соответствует целям и видению всей системы.

Для нашей модели «От архитектуры до релиза» мы определили только один экземпляр команды разработки платформы. Однако вам может потребоваться

развернуть несколько таких команд, чтобы сохранить управляемость ими. В таком случае вам также необходимо будет подумать о том, как несколько команд разработки платформы будут координировать свои действия и предлагать услуги остальной части организации.

Команды поддержки и разработки сложных подсистем

Трех созданных команд достаточно для развертывания микросервисной системы. Но помимо описанных основных потребностей могут быть дополнительные, обусловленные, например, наличием набора важных или сложных системных особенностей, которым понадобится специальная команда.

В нашей модели микросервисов мы решили создать дополнительную, специализированную команду выпуска. Она несет ответственность за выпуск (или развертывание) микросервисов в производственной среде. В общем случае команда по работе с микросервисами может самостоятельно развертывать свои сервисы непосредственно в производственной среде, но, по нашему опыту, так происходит не всегда.

Такое положение связано с тем, что в большинстве организаций обычно проводятся дополнительное тестирование и приемка перед выпуском сервиса в работу. Поэтому вместо развертывания в рабочей среде команды микросервисов предоставляют собранный и протестированный контейнер. Затем он автоматически развертывается командой выпуска, которая координирует работу по тестированию, утверждению и развертыванию изменений.

Команда выпуска относится к типу команд разработки сложных подсистем. Ее члены обладают специальными знаниями о процессе выпуска, утверждения и развертывания и сотрудничают с потоковыми командами для выполнения этой работы. Чтобы задокументировать структуру команды выпуска, создайте файл `release-team.md` со следующими свойствами:

```
# Команда выпуска

## Тип команды
Команда разработки сложной подсистемы

## Размер команды
5-8 человек

## Отвечает за
* выпуск микросервисов в работу
* координацию утверждений для выпусков
```

Далее мы добавим команду выпуска в диаграмму топологии нашей команды. Команды разработки сложных подсистем изображаются своим цветом.

Итак, откройте схему топологии команды и добавьте квадрат (мы выбрали для него красный цвет) в конце поля команды микросервисов, как показано на рис. 2.4.

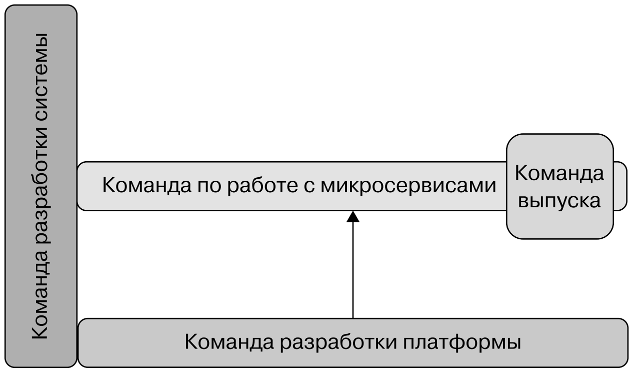


Рис. 2.4. Команда выпуска

Как мы можем видеть в новой топологии, одним из компромиссов с введением команды выпуска являются связанные с этим затраты на координацию. Для крупных масштабных систем это может стать большой проблемой. Например, если понадобится ежедневно выпускать новые версии нескольких микросервисов, то команде выпуска будет сложно координировать всю эту деятельность. Оказавшись в такой ситуации, вы должны будете изменить структуру команды и переложить ответственность за развертывание на отдельные команды микросервисов.

Команда выпуска — последняя, лежащая в основе модели микросервисов. Но, чтобы закончить структуру, необходимо рассмотреть команды, которые будут использовать наши микросервисы. Рассмотрим этот аспект далее.

Команды потребителей

Микросервисы приносят пользу, только если они используются. Поэтому стоит определить потребителей наших микросервисов и то, как они будут взаимодействовать с командами системы. В некоторых архитектурах в этот процесс могут вовлекаться команды разработчиков мобильных и веб-приложений или даже сторонние организации. В нашей модели основным потребителем системы микросервисов является команда API.

Команда API отвечает за предоставление доступа к нашим микросервисам другим командам разработчиков посредством прикладного программного интерфейса (application programming interface, API). Например, команда разработчиков мобильных приложений будет взаимодействовать с API, выпущенным этой командой, и никогда не будет напрямую обращаться к нашим микросервисам.

Мы углубимся в детали API и архитектуры несколько позже, а пока определим свойства команды API и ее обязанности. Для этого создадим файл `api-team.md` и заполним его следующим образом:

```
# Команда API

## Тип команды
Потоковая

## Размер команды
5-8 человек

## Отвечает за
* проектирование, разработку и поддержку API на границе системы
* подключение API к внутренним микросервисам
```

Как и команда по работе с микросервисами, команда API является потоковой. Это связано с тем, что ей необходимо постоянно вносить в API изменения, отражающие потребности бизнеса и запросы потребителей. Особый нюанс команды API заключается в том, что, поскольку API должен вызывать микросервисы для функционирования, он зависит от команды микросервисов.

Можно смоделировать эти свойства взаимодействия в нашей модели топологии команды, добавив в верхнюю часть диаграммы еще один прямоугольник для представления команды API. Он должен быть того же цвета, что и команда по работе с микросервисами (мы использовали желтый), так как это тоже потоковая команда. Чтобы отразить зависимость между командами микросервисов и API, мы снова используем черную стрелку, чтобы показать модель взаимодействия XaaS. Она указывает, что команда микросервисов должна убедиться в доступности их сервисов для вызова и использования в режиме самообслуживания.

Когда вы закончите, диаграмма должна выглядеть примерно так, как показано на рис. 2.5.

После добавления команды API наша топология стала очень похожа на готовый продукт, который был показан в начале этого раздела. Определив топологию, мы можем видеть расположение основных точек координации, в которых выполняется работа. В целом наша модель обеспечивает достаточно независимый,

автономный способ работы. Однако нам нужно будет потратить некоторое время и силы на создание облачной платформы как услуги, чтобы заставить модель работать.

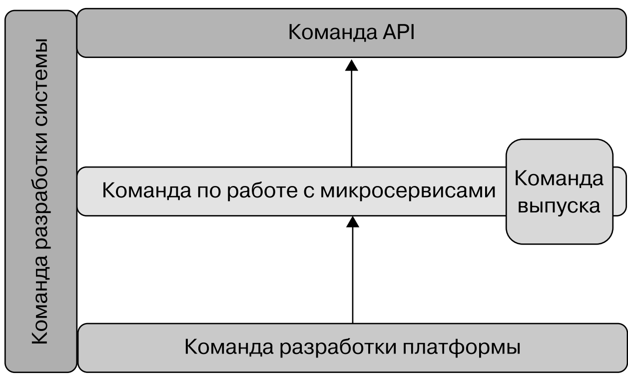


Рис. 2.5. Итоговая топология команд, включающая команду API

Определив базовую топологию команд, мы можем увидеть, как эта работа связана с целью нашей главы — построением операционной модели.

Резюме

В совокупности решения, определения команд и топологии, которые мы только что создали, формируют нашу операционную модель микросервисов. С ее помощью мы определили, какие команды необходимо создать, их характеристики и обязанности, а также ожидаемые взаимодействия между ними. Это важный этап проектирования, который повлияет на остальную работу над микросервисами. Фактически каждое решение, которое мы примем с этого момента, будет в значительной степени зависеть от только что созданной операционной модели.

По правде говоря, мы не очень углублялись в разработку операционной модели. На практике имеет смысл нарисовать несколько схем топологии команд, чтобы отразить различные типы режимов взаимодействий. Например, задачи устранения неполадок в нашей системе, вероятно, потребуют иной модели взаимодействия, отличной от уже созданной. Аналогичным образом мы не составили диаграмму взаимодействий, необходимых для изменения услуг, предоставляемых облачной платформой, командами разработки системы и выпуска.

Помимо создания первоначальной структуры операционная модель должна постоянно совершенствоваться. Одна из приятных особенностей представле-

ния определений и топологий нашей команды в виде документов заключается в возможности обращаться с ними как с кодом. То есть мы можем изменять версии и управлять изменениями по мере развития системы. Возможно, вы даже захотите добавить дополнительные ресурсы проектирования в свою коллекцию. Например, можно определить соглашения об уровне обслуживания для команд разработки платформы и списки навыков для потоковых команд.

Конечной целью в этой главе было создание основы для дальнейшего проектирования и разработки. Наш упрощенный подход к топологии и структуре команды помог добиться этого. Имея на руках операционную модель, можно приступать к разработке реальных микросервисов. Этот вопрос мы рассмотрим в главе 3.

ГЛАВА 3

Разработка микросервисов: процесс SEED(S)

Если вы помните, в главе 1 утверждалось, что основным преимуществом внедрения архитектуры микросервисов является возможность увеличить скорость разработки крупномасштабных систем без ущерба для их безопасности. Это чрезвычайно важное преимущество для организаций, решающих сложные задачи. Обратите внимание, что это результат сознательного замысла, а не случайность. Во всех ситуациях, кроме самых простых, невозможно получить успешную архитектуру микросервисов, не занимаясь эффективным и четким сквозным проектированием системы.

В данной главе мы представим эволюционный процесс проектирования микросервисов. Эта методология была впервые сформулирована одним из авторов в медицинском стартапе, соучредителем которого он являлся, а затем успешно внедрена в многочисленных проектах других компаний. Гибкий подход оказался столь же эффективным для небольших организаций, решающих сложные задачи, например для новаторского стартапа, совершившего революцию в сфере здравоохранения, и для крупной организации с тысячами инженеров-программистов в сотнях команд.

КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ СТАНДАРТНЫЙ ПРОЦЕСС ДЛЯ ПРОЕКТИРОВАНИЯ СЕРВИСА

Используйте стандартный, повторяемый процесс, чтобы создать высококачественный, ориентированный на клиента проект сервиса в вашей системе.

Система проектирования микросервисов, описанная в этой главе, представляет собой нисходящую многоступенчатую методологию и набор повторно используемых процессов, где каждый последующий шаг вытекает из предыдущего. В силу ее эволюционной природы мы называем систему так: *семь основных*

этапов эволюционного проектирования сервисов (Seven Essential Evolutions of Design for Services, SEED(S)). Мы находим это ироничное название подходящим, учитывая, что результаты анализы, выполняемых с помощью этой методологии, часто оказываются необходимыми семенами (seeds), из которых вырастает красивая разветвленая система микросервисов. Подобно тому как прекрасный цветущий сад начинается с посадки нескольких семян, процесс анализа и проектирования SEED(S) является важным первым шагом на пути внедрения микросервисов, облегчающим последующие этапы, связанные с программированием.

Семь основных этапов эволюционного проектирования сервисов: метод SEED(S)

Как отмечают Джеймс Льюис и Мартин Фаулер в своей основополагающей статье о микросервисах, одной из основных черт архитектуры микросервисов является разделение системы на компоненты с помощью сервисов (https://oreil.ly/DVxRp). Под сервисами авторы подразумевают программные компоненты, которые могут развертываться независимо и быть доступны по стандартным сетевым протоколам, таким как запрос веб-сервиса или вызов удаленной процедуры. Организуя компоненты системы в виде сервисов, среди прочего мы обязуемся определять для них явные общедоступные интерфейсы. Повышение гибкости и удобства использования этих интерфейсов за счет хорошей организации может положительно сказаться на надежности архитектуры системы и продуктивности разработчиков.

Процесс SEED(S) обеспечивает повторяемую и проверенную на практике методологию разработки удобных и надежных сервисных интерфейсов.

Следует также отметить, что в качестве общего подхода методология SEED(S) может применяться для разработки API не только микросервисов, но и многих других типов сервисов, включая RESTful и GraphQL, создаваемых для поддержки пользовательских интерфейсов. Такой широкий диапазон применимости не должен вызывать удивления. В конце концов, с технической точки зрения микросервис тоже своего рода API, просто разработанный с учетом определенных ограничений, минимизирующих потребности в координации.

Итак, представляем семь этапов процесса SEED(S).

1. Идентификация участников.
2. Определение заданий, которые должны выполнять участники.
3. Выявление шаблонов взаимодействия с помощью диаграмм последовательностей.

4. Выделение высокоуровневых действий и запросов на основе заданий, которые необходимо выполнить (jobs to be done, JTBD), и шаблонов взаимодействий.
5. Описание каждого запроса и действия в виде спецификации с использованием открытого стандарта (такого как спецификация OpenAPI (OAS)) или схемы GraphQL.
6. Получение обратной связи по спецификации API.
7. Реализация микросервисов.

Рассмотрим каждый из этих этапов подробнее и обдумаем, как можно их применить при проектировании сервисов.

Идентификация участников

SEED(S) не только эволюционная методология, но и четко ориентированный на клиента подход, рассматривающий проектируемые сервисы как продукты. К настоящему времени мантра «API — это продукты» не слишком нова. Мы кричали о ней (https://oreil.ly/y8CHg) на всех перекрестках в течение многих лет. Самое замечательное, что взгляд на API и сервисы как на продукты позволяет нам повторно использовать множество методов из делового мира, где в этом нет ничего нового. На самом деле наука и искусство управления продуктами появились значительно раньше, чем API и даже сам Интернет. Многие считают, что управление продуктами как область деятельности (https://oreil.ly/lunbc) зародилось еще в 1930-х годах в виде первых попыток Procter & Gamble и Нила Макэлроя (https://oreil.ly/yYLlg) повысить продажи мыла марки Camay от P&G. В последующие десятилетия управление продуктами значительно эволюционировало, и мы извлекли много уроков, которые можно повторно использовать в гораздо более молодом пространстве управления API/сервисами. Если API являются продуктами, то мы должны иметь возможность применять аналогичные методы для разработки API, которые используются в управлении продуктами.

При разработке продукта и, следовательно, API или сервиса мы должны понимать клиента. Для кого предназначен сервис? Как правило, в области API и управления сервисами мы не называем этих персон «клиентами», а используем менее употребимое в коммерции слово «участник» (или «актор»), устраняя любую случайную, непреднамеренную коннотацию финансовой сделки или выгоды, присутствующей между потребителем и издателем сервиса.

Использование термина «участник» на первом этапе моделирования методологии SEED(S) заимствовано из области проектирования взаимодействий, где для тех же целей используется термин «пользовательская персона» (user personas). Понятие персон как инструмента проектирования взаимодействий ввел

Алан Купер в своей книге 1998 года *The Inmates Are Running the Asylum*[1] (Sams Publishing) (https://learning.oreilly.com/library/view/inmates-are-running/0672326140), и с тех пор оно получило широкое распространение. Однако следует признать, что с тех пор понятие персоны получило свою долю критики (https://oreil.ly/g1SYg) (а какая методология обошлась без этого?), и некоторые команды по работе с продуктами активно выступают за использование вместо этого реальных пользовательских данных. Обсуждение плюсов и минусов персон в управлении продуктами выходит далеко за рамки этой книги. Появление участников вдохновлено персонами пользователей, но это не идентичные понятия. Цель участников — помочь в моделировании на этапе процесса проектирования, когда фактические пользовательские данные обычно ограниченны.

Основной мотивацией начинать моделирование с определения участников является помощь в установлении масштабов задач и расстановке приоритетов. Типичные недостатки проектирования API и сервисов в нашей отрасли — чрезмерная абстракция и отсутствие ясности в отношении потребностей пользователей. Многие API просто открывают доступ к некоторым таблицам базы данных по протоколу HTTP или пытаются предоставить прямой сетевой доступ к внутренним компонентам приложения через вызовы удаленных процедур (remote procedure calls, RPC). С такими подходами удовлетворять потребности клиентов и достигать бизнес-целей весьма сложно. Это не должно вызывать удивления. Если не спрашивать себя: «Кто будет использовать этот API?» и «Каковы их потребности?» — то как можно разработать решения, отвечающие их потребностям? И все же слишком многие API и сервисы спроектированы именно так: с учетом целей издателя сервиса, а не потребителя. SEED(S) решает эту перевернутую проблему, требуя идентифицировать участников на самом первом шаге.

КЛЮЧЕВОЕ РЕШЕНИЕ: ОПРЕДЕЛИТЕ ОБЛАСТЬ, ОХВАТЫВАЕМУЮ ПРОЕКТИРУЕМЫМ СЕРВИСОМ, ИДЕНТИФИЦИРОВАВ КЛЮЧЕВЫХ УЧАСТНИКОВ

Начните проектирование сервисов с определения ключевых участников в вашей области, чтобы добиться полного охвата потребностей клиентов.

Существует несколько основных правил для определения правильного набора участников для ваших задач.

1. Как и в случае с пользовательскими персонами Купера, каждый участник должен быть больше *конкретным*, чем *точным*. Под этим мы подразумеваем, что определение границ ключевых черт, отличающих различных участников нашего проекта, важнее точного определения того, кто эти участники. Мы всегда должны помнить, что на этапе моделирования любая созданная

[1] *Купер А.* Психбольница в руках пациентов. — СПб.: Питер, 2018.

модель по определению неточна: дело не в том, что мы не можем уловить детали, а скорее в том, что мы не заботимся о каждой отдельной детали и пытаемся уловить приоритетное представление реальности, актуальное для нас.

2. Пересекающиеся или слишком широкие определения участников обычно настораживают. Участники также должны определяться с учетом контекста. Наличие общекорпоративного «портфолио» участников, использующихся при проектировании всех приложений, — это больше, чем признак проблемы: это сигнал «все сигналы тревоги включены, звоните 911» и верный признак того, что процесс был сорван и скомпрометирован.
3. Модели определения участников в первую очередь должны представлять потребности, болевые точки и особенности поведения, присущие каждому архетипу участника. Эти потребности и особенности поведения отличают один тип участников от другого, являются классифицирующими признаками и должны иметь как можно меньше совпадений.
4. Чем меньше, тем лучше — используйте как можно меньше отдельных участников для описания вашей предметной области, но не меньше, чем необходимо. В большинстве случаев наличие более пяти участников для сервиса можно рассматривать как признак неправильного определения приоритетов или слишком широких границ сервиса.

Примеры участников в нашем учебном проекте

Ниже приведены некоторые из возможных участников для нашего учебного проекта, представленного в главе 1: системы онлайн-бронирования авиакомпании или, более конкретно, ее подсистемы бронирования авиабилетов.

- *Постоянный пассажир*. Эмма путешествует по работе, имеет элитный статус лояльности к авиакомпании, управляет своими поездками через свою рабочую систему бронирования и использует ряд взаимосвязанных приложений, чтобы не отставать от своего плотного графика. Благодаря своему статусу лояльности она имеет право на множество льгот. Часто планируя поездки в сжатые сроки, путешествуя с семьей, она использует мили лояльности.
- *Семейный отдыхающий*. Райли и его супруга в основном путешествуют со своими детьми во время отпуска. Обычно они планируют поездки заранее и путешествуют нечасто.
- *Агент по обслуживанию клиентов авиакомпании*. Шон — опытный агент по обслуживанию клиентов, помогающий путешественникам с бронированием, перебронированием и решением проблем во время поездки и после нее по телефону и в онлайн-чате.

Идентифицировав участников для нашего проекта, мы сможем проанализировать действия, которые они будут выполнять с помощью нашей системы. Что под этим подразумевается, вы узнаете в следующем разделе.

Определение действий, выполняемых участниками

Определив целевой класс клиентов (в нашем случае участников), нужно потратить значительное количество времени, чтобы понять, какие задачи они будут решать. И только после этого мы сможем создать решение, максимально удовлетворяющее их потребности. Это критический момент, который часто неправильно понимают или игнорируют при разработке сервисов и API, поэтому попытаемся обосновать его важность.

Любая эффективная методология проектирования API или сервисов, включая SEED(S), основана на фундаментальной предпосылке, упоминавшейся ранее: API и микросервисы — это продукты, и при их разработке можно успешно использовать богатый набор инструментов управления продуктами, который разрабатывался на протяжении многих десятилетий. Мы уже применили один такой инструмент к нашему процессу моделирования: идентификацию участников на манер пользовательских персон в модели взаимодействий. На этом втором этапе мы еще глубже погрузимся в разработку продукта, поэтому, возможно, стоит повторить вопрос: почему мы считаем, что API — это продукты? В конце концов, возможности, предоставляемые по сети с использованием стандартных протоколов, то есть то, что мы называем API, не обязательно имеют очевидное сходство с мылом для рук, зимними куртками, смартфонами и другими привычными материальными продуктами.

И вообще, существует ли общее определение *продукта*? Такого определения, о котором было бы всем известно, нет, поэтому приведем цитату из Википедии (https://oreil.ly/blWDL):

> «Мы определяем продукт как все, что может быть предложено рынку для привлечения внимания, приобретения, использования или потребления, что может удовлетворить желание или потребность. Продукты — это нечто большее, чем просто материальные объекты, такие как автомобили, компьютеры или мобильные телефоны. В широком смысле к “продуктам” относятся также услуги, события, люди, места, организации и идеи или их сочетания».
>
> *Котлер и др. Principles of Marketing[1], 7th edition (Pearson)*

[1] *Котлер Ф., Армстронг Г.* Основы маркетинга.

Сервисы, будь то веб-API или микросервисы, действительно подходят под это определение: производители предлагают своим потребителям услуги, удовлетворяющие их нужды, и это соотношение спроса и предложения может создать «рынок».

Потребителями API обычно являются интерфейсные (веб-, мобильные) или сторонние (партнерские) приложения, в то время как в роли потребителей микросервисов выступают различные части самой системы, но это различие не очень существенно по отношению к процессу их проектирования. Мы углубимся в определение различий между API и микросервисами позже в этой главе.

Итак, если API и микросервисы *являются* продуктами, то как можно создавать лучшие их образцы? Идентификация участников — первый шаг, но что дальше? Они должны решать проблемы клиента. Увы, печальная реальность такова, что слишком много продуктов разрабатываются с точки зрения поставщика решений, зацикленного на том, что он может предложить, вместо того чтобы сосредоточиться на проблемах, которые хотят решить клиенты. Вероятно, наиболее кратко объяснить эту проблему можно с помощью знаменитых слов профессора маркетинга Гарвардской школы бизнеса Теодора Левитта: «Люди не хотят покупать сверло диаметром в четверть дюйма. Они хотят просверлить отверстие в четверть дюйма!»[1] И действительно, если вы производите сверла, то должны понимать, что реальная задача, которую пытаются решить клиенты, может заключаться в том, чтобы повесить картину на стену, а не купить самое совершенное сверло. Если вы не осознаете этого, продолжая совершенствовать сверло, то в конце концов вас переиграет изобретатель, который придумает более простое альтернативное решение для получения четвертьдюймовых отверстий в стенах. Это может быть химическая реакция или что-то еще — мы не знаем, — но это произойдет.

Как показывает история технического прогресса, проблемы неподвластны времени: решения постоянно меняются и развиваются. Показательный пример — никто больше не использует магнитные ленты или гибкие диски для хранения данных, но необходимость сохранения и транспортировки данных не исчезла, даже притом, что теперь в основе лежит облачное хранилище. Новаторы должны больше концентрироваться на решении проблем и меньше — на совершенствовании инструментов, которые обычно являются временными.

[1] Цитата из: *Кристенсен К. М. и др.* Чего хотят клиенты от ваших продуктов. — Гарвардская школа бизнеса, 2016. https://oreil.ly/NKolz.

Профессор Гарвардской школы бизнеса Клейтон Кристенсен называет это наблюдение «теорией работы, которую нужно выполнить», объясняя, что:

> «[клиенты] часто покупают вещи, поскольку сталкиваются с проблемой, которую хотели бы решить».
>
> Клейтон Кристенсен. *The Innovator's Solution*[1] (Harvard Business Review Press)

В статье Гарвардской школы бизнеса «Чего хотят клиенты от ваших продуктов», процитированной выше, Кристенсен и его соавторы объясняют, что проект продукта успешен и клиенты находят его желательным, когда «работа, а не клиент является фундаментальной единицей анализа».

КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ ЗАДАНИЯ В КАЧЕСТВЕ ЕДИНИЦЫ АНАЛИЗА

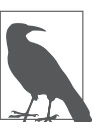
Используйте задания, которые должны выполнить ключевые участники в вашей предметной области, в качестве единицы анализа для сбора требований.

Использование формата истории заданий для фиксации JTBD

Для каждого из идентифицированных участников необходимо найти основные задания, которые нужно выполнить (JTBD). Для единообразия, а также чтобы убедиться, что ключевые точки данных хорошо документированы, фиксируем JTBD в стандартном формате. Процесс SEED(S) использует формат *истории заданий*, определенный Полом Адамсом (https://oreil.ly/VJQui): «когда <обстоятельство>, мне хотелось бы <мотивация>, так что я могу <цель>» (рис. 3.1).

Рис. 3.1. Структура формата истории заданий

История заданий сосредоточена на обстоятельствах, мотивации участника для выполнения работы и цели, которую он пытается достичь.



Обратите внимание, что в оригинальном формате Адамса истории заданий записываются от первого лица. В SEED(S), как вы увидите далее в этой главе, мы предпочитаем записывать истории заданий от третьего лица, подчеркивая, кто является фактическим действующим лицом в истории.

[1] *Кристенсен К.* Дилемма инноватора.

Если вы знакомы с пользовательскими историями (https://oreil.ly/PT6-v) по практике применения Scrum или других методологий гибкой разработки, то, возможно, заметили, что история заданий выглядит почти идентично. Однако, как объясняет Алан Клемент в своем сообщении в блоге «Замена пользовательских историй на истории заданий» (https://oreil.ly/UGXy_), между ними есть принципиальные различия. Пользовательские истории вращаются вокруг персоны пользователя. Они начинаются с «как <персона>», тогда как истории заданий не обращают внимания на персону и вместо этого подчеркивают обстоятельства.

Это важно и согласуется с утверждением Кристенсена: «Задача, а не клиент является фундаментальной единицей анализа». Это так же верно еще и потому, что в контексте описания конкретной работы персона не имеет значения. Если мне нужно повесить картину на стену, то на самом деле не имеет значения, являюсь ли я лицензированным подрядчиком или начинающим домовладельцем. Мне нужно просверлить в стене отверстие (или несколько) в четверть дюйма. Важен контекст, обстоятельства, в которых у нас есть мотивация для достижения цели, а не кто мы такие есть. Проще говоря, мы определяем участников, чтобы расширить список заданий, но в момент описания каждого задания для этого участника нужно определить обстоятельства, а не просто повторять десять раз «как постоянный пассажир...»

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ СТАНДАРТНЫЙ ФОРМАТ ИСТОРИЙ ЗАДАНИЙ

Применяйте стандартный формат для записи JTBD (известный как Job Story, или история заданий), чтобы единообразно фиксировать обстоятельства, мотивы и цели для всех ваших заданий.

Примеры JTBD в нашем проекте

Выберем несколько JTBD для семейного отдыхающего участника.

1. *Когда* Райли планирует перелет для отдыха всей семьей, *ему хотелось бы* иметь возможность отфильтровать доступные рейсы по нескольким критериям, в том числе: доступность четырех смежных мест, количество пересадок, пересадки в аэропортах, где есть удобства для маленьких детей, и т. д., *чтобы семья могла* совершить перелет с максимальным комфортом.
2. *Когда* Райли планирует в спешке внеочередной отдых всей семьей на долгие выходные, *ему хотелось бы* получить интересные предложения поездок, доступных по цене и с коротким перелетом, *чтобы получить* список вариантов для рассмотрения.

А теперь разберем некоторые задания для постоянных пассажиров-участников.

1. *Когда* планы Эммы меняются и она не может лететь ранее забронированным рейсом, *ей хотелось бы* без затруднений перенести свой полет, *чтобы* получить билет на рейс, соответствующий ее новым планам.
2. *Когда* Эмме не нравится место, предложенное в данный момент, *ей хотелось бы* выбрать альтернативное место, *чтобы* получить больше удовольствия от полета.

Наконец, JTBD для агента по обслуживанию клиентов может выглядеть следующим образом.

1. *Когда* клиент звонит Шону, Шону *хотелось бы* получить заявку на обслуживание, предварительно заполненную информацией о клиенте, *чтобы он мог* начать отслеживать прогресс в решении потребности клиента.
2. *Когда* клиент просит Шона подобрать ему удобный рейс для поездки, Шону *хотелось бы* иметь возможность найти подходящий рейс, используя гибкий набор критериев фильтрации, *чтобы* удовлетворить потребности клиента и забронировать рейс.

Когда это возможно, всегда стоит извлекать истории заданий из анализа пользователей. А для последовательного изложения результатов исследования можно с успехом использовать простой, нетехнический и согласованный формат.

Истории заданий могут служить отличной основой для бесед с экспертами по предмету и реальными клиентами, но они неудобны для получения фактических технических требований. Их лучше всего преобразовать в более удобный для разработчиков формат, чему и посвящены следующие несколько этапов процесса SEED(S).

Выявление шаблонов взаимодействия с помощью диаграмм последовательностей

Задания обычно описываются менеджерами по продуктам с точки зрения ценности для бизнеса и редко каким-либо образом соответствуют нашим целевым сервисам. Чтобы продолжить проектирование, нам нужно понять шаблоны взаимодействия сервисов в нашей предметной области, которой принадлежат эти сервисы. Для сложных взаимодействий линейный список историй заданий не сможет в достаточной степени поддержать усилия по проектированию. Вместо

этого лучше нарисовать диаграмму взаимодействий, объясняющую последовательность событий в модели.

В духе повторного использования существующих стандартов SEED(S) рекомендует применять диаграммы последовательностей на унифицированном языке моделирования (Unified Modeling Language, UML) для этой задачи. Вы можете использовать любой другой подход к построению диаграмм для выражения своей модели, поскольку цель состоит лишь в том, чтобы передать намерение и модель. Однако если вы решите задействовать диаграммы последовательностей UML, то мы настоятельно рекомендуем использовать один из форматов на основе языка Markdown, такой как PlantUML (http://plantuml.com).

Мы рекомендуем такой подход, потому что моделирование микросервисов — командная работа. Использование текстового формата вместо графического позволит членам команды:

- отделить моделирование от личного выбора редактора. PlantUML и другие подобные форматы можно редактировать во множестве различных редакторов. Например, формат PlantUML поддерживается в Confluence от Atlassian, ПО для управления знаниями, используемом многими командами разработчиков программного обеспечения. Существуют также бесплатные онлайн-редакторы, такие как LiveUML (https://liveuml.com) и PlantText (https://oreil.ly/VpjNq);
- просто и эффективно управлять версиями диаграмм. Текстовые файлы легко исследовать, объединять и просматривать в запросах на извлечение, в отличие от двоичных графических файлов;
- удобно интегрировать моделирование в процесс выпуска. Диаграммы становятся кодом, и все, что можно сделать с кодом, можно сделать и с диаграммами. Например, их можно хранить в системе управления версиями, такой как Git.

КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ ДИАГРАММЫ ПОСЛЕДОВАТЕЛЬНОСТЕЙ PLANTUML ДЛЯ ВЫЯВЛЕНИЯ ШАБЛОНОВ ВЗАИМОДЕЙСТВИЙ

Чтобы выявить шаблоны взаимодействий в методологии SEED(S), решено использовать диаграммы последовательностей UML, выраженные в текстовом формате (Markdown), таком как PlantUML.

В данный момент мы находимся на этапе разработки технической модели, отражающей требования, собранные в физическом мире. Истории заданий и участники отражают требования физического мира. Как правило, они не

соответствуют напрямую техническим взаимодействиям. Поэтому в вашей модели взаимодействий события не обязательно должны происходить между участниками, описанными на первом этапе процесса SEED(S). Они также не должны напрямую соответствовать заданиям. Скорее, ваши диаграммы взаимодействий могут быть более глубокими и показывать, как требования пользователей преобразуются во взаимодействия между сервисами на техническом уровне.

Например, вот как может выглядеть очень простая диаграмма, описывающая взаимодействия, связанные с заданиями, которые мы определили ранее в этой главе:

```
@startuml

actor Агент
participant "Агентское обслуживание" as AS
participant "API резервирования" as rAPI
participant "reservationCRUD " as rCRUD

AS -> rAPI: checkRes(reservationId)
rAPI -> rCRUD: reserve(data)
rAPI -> rCRUD: cancel(reservationId)

@enduml
```

В LiveUML эта диаграмма отображается, как показано на рис. 3.2.

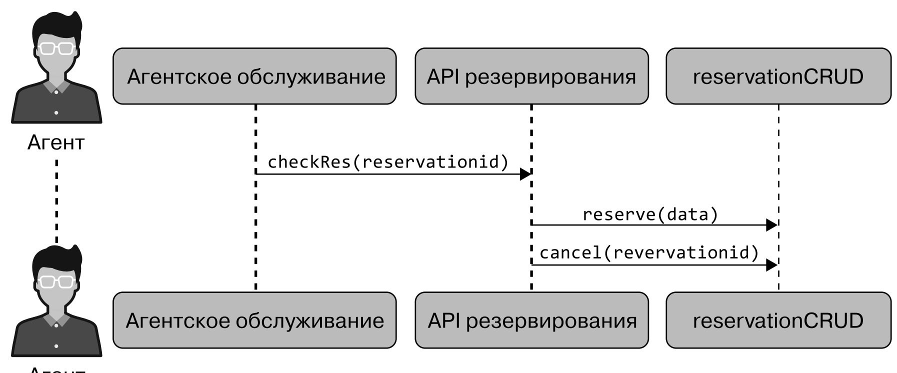


Рис. 3.2. Пример отображения последовательности UML в PlantUML

Здесь агентское облуживание — это пользовательский интерфейс (веб- или мобильное приложение), который агенты могут использовать напрямую, API резервирования — это REST или GraphQL API, вызывающий приложение,

а reservationCRUD — один из микросервисов, который подпитывает указанный API.

Создав диаграммы последовательностей взаимодействий, мы сможем зафиксировать технические требования к микросервису или API в виде набора действий и запросов, описанных с использованием стандартного синтаксиса. Рассмотрим в следующем разделе, что это такое и как они выглядят.

Выделение действий и запросов из JTBD

Истории заданий дают отличную основу для обсуждений с экспертами в данной области и получения информации о потребностях клиентов. Однако они часто малопригодны для начала фактического проектирования спецификаций API. Но, проанализировав шаблоны взаимодействий с сервисами и получив возможность их визуализировать, мы сможем преобразовать задания в более технически ориентированные интерфейсные контракты и значительно упростить процесс проектирования. Следуя принципу разделения на команды и запросы (command query separation, CQS) Бертрана Мейера[1], в SEED(S) мы моделируем контракты интерфейса как наборы двух различных типов взаимодействий: действий (команды в CQS) и запросов.

КЛЮЧЕВОЕ РЕШЕНИЕ: РАЗДЕЛИТЬ КОНЕЧНЫЕ ТОЧКИ СЕРВИСА НА КОМАНДЫ И ЗАПРОСЫ

Используйте принцип CQS для моделирования стороны действия сервисов отдельно от стороны запроса и документирования каждой из них в их собственном стандартном формате.

В SEED(S) в роли запросов выступают поисковые запросы с определенными входными и выходными данными. Они должны быть четко сформулированными контрактами между клиентом и сервером: какие входные данные отправляет клиент и какой ответ ожидает. Запросы явно отличаются от действий тем, что не изменяют состояния системы («не имеют побочных эффектов»).

Действия, напротив, — это запросы, вызывающие какое-то изменение состояния — они не только имеют побочные эффекты, но и вся их цель состоит в том, чтобы вызвать их. Как и запросы, действия тоже имеют четко определенные контракты — для входных данных, ожидаемых результатов и ожидаемых ответов.

[1] *Meyer B.* Object-Oriented Software Construction. 2nd ed. — N. Y.: Prentice Hall, 2000. (*Мейер Б.* Объектно-ориентированное конструирование программных систем.)

Как и в случае с историями заданий, мы рекомендуем использовать стандартный формат для описания запросов и действий. Вот примерный шаблон описания запросов.

- Выразительное описание запроса.
  - Ввод: список входных переменных.
  - Ответ: список элементов выходных данных.

Аналогично выглядит стандартизированный формат описания действий.

- Выразительное описание действия.
  - Ввод: список входных переменных.
  - Ожидаемый результат: описание вызванного побочного эффекта.
  - Ответ (не обязательно): список элементов данных в ответе (если таковые имеются).

Обратите внимание, что истории заданий не всегда содержат ровно один запрос или действие. Историю заданий можно преобразовать в несколько запросов и действий, и полученный запрос или действие могут объединять несколько историй заданий. SEED(S) — это процесс моделирования, проектирования и открытий, а не роботизированный процесс, созревший для устранения фактора человеческого суждения.

Примеры запросов и действий для нашего проекта

Рассмотрим несколько примеров преобразования наших историй заданий в набор запросов и действий.

Запросы

В одной из наших историй заданий описан участник (Райли), предпочитающий семейный отдых и желающий найти рейс, который будет соответствовать требованиям его семьи к комфорту в путешествии. Для этого он указывает подробные предпочтения, такие как количество смежных мест, максимальное количество пересадок и т. д. Чтобы удовлетворить эти потребности, нужен контракт запроса, который позволит указать все такие предпочтения в качестве входных данных для поискового запроса. Следовательно, наше определение запроса может выглядеть так.

Запрос 1. Поиск рейса

- Ввод: `departure_date`, `return_date`, `origin_airport`, `destination_airport`, `number_of_passengers`, `baby_friendly_connections`, `adjacent_seats`,

max_connections, minimum_connection_time, max_connection_time, order_criteria [object], customer_id (дополнительно; проверить привилегии лояльности).

- Ответ: список рейсов, удовлетворяющих критериям.

В другой нашей истории задания описывалось обстоятельство, при котором планы постоянного путешественника внезапно меняются и он не может отправиться в ранее запланированную дату/рейс. Этому участнику необходимо перенести свою бронь. Чтобы выполнить эту задачу, можно представить минимальную требуемую информацию:

- уникальный идентификатор предыдущего бронирования, чтобы получить данные об аэропортах отправления и назначения, а также любые другие предпочтения, чтобы появилась возможность автоматически задать их для нового поиска, не требуя от путешественника повторного ввода;
- новые дату вылета и дату возвращения, которые подходят путешественнику.

Выполнив поиск, мы должны получить список рейсов, соответствующих новым датам, чтобы представить их клиенту и позволить ему выбрать, на какой рейс тот хотел бы перебронировать свой билет.

Основываясь на этом анализе, можно сделать вывод, что спецификация запроса «перебронирование» может выглядеть следующим образом.

Запрос 2. Поиск альтернативных рейсов для измененной даты

- Ввод: `reservation_id`, `new_departure_date`, `new_return_date`.
- Ответ: список альтернативных рейсов.

Действия

Выполнив анализ, аналогичный тому, что использовался для выделения запросов из историй заданий, можно выделить действия, необходимые для перебронирования и изменения места в салоне самолета.

Перебронирование поездок

- Ввод: `original_reservation_id`, `new_flight_id`, `seat_ids[]`.
- Ожидаемый результат: бронирование места на новом рейсе или возврат ошибки; если новый рейс успешно забронирован, то старая бронь отменяется.
- Ответ: код успеха или объект с полной информацией об ошибке.

Смена посадочного места

- Ввод: `reservation_id`, `customer_id`, `requested_seat_ids[]`.
- Ожидаемый результат: зарезервировано новое место, если доступно и пассажир соответствует требованиям; если новое место успешно зарезервировано, то бронь старого места отменяется.
- Ответ: код успеха или объект с полной информацией об ошибке.

В некоторых сложных случаях можно обнаружить, что подхода на основе действий и запросов к определению интерфейсных контрактов может оказаться недостаточно. В этих случаях, чтобы учесть более сложные требования, мы настоятельно рекомендуем использовать Microservice Design Canvas (https://oreil.ly/tIxEi) Мэтта Макларти (https://oreil.ly/lLRpj). Канва проектирования и «анализ действий и запросов» являются взаимозаменяемыми методами на одном и том же этапе процесса SEED(S). Канва — более мощный инструмент; мы не рассматриваем его в этой книге, но с ним стоит познакомиться.

Выделив набор действий и запросов или канву проектирования микросервиса, мы можем преобразовать их в формальную спецификацию интерфейса.

Описание каждого запроса и действия в виде спецификации с использованием открытого стандарта

Как правило, важно формально описать контракт интерфейса API или микросервиса, прежде чем начать воплощать его в код. Такие формальные описания контрактов служат взаимно согласованным договором между производителем сервисов и потребителями или разработчиками клиентов API. Контракты также легко преобразуются в удобную для пользователя документацию и интерактивные интерфейсы. Контракты, реализованные с использованием открытых стандартов, таких как OpenAPI Spec (https://www.openapis.org) и GraphQL (https://graphql.org), широко поддерживаются богатым набором инструментов, позволяющих отображать документацию, упрощающих создание порталов разработчиков и т. д.

В этом разделе мы возьмем определение действия, которое описали на предыдущем этапе SEED(S), и разработаем для него конечную точку RESTful, используя Open API Specification (OAS) (https://oreil.ly/JoiGg).

Спецификация OAS описывает RESTful API в стандартной манере, не зависящей от технической среды. Она определяется OpenAPI Initiative, совместным

проектом Linux Foundation. На момент написания книги последней была версия OAS 3.0.2 (https://oreil.ly/c9U2V).

Связь с микросервисами не обязательно должна осуществляться посредством RESTful API. Также часто используются GraphQL (https://oreil.ly/C1c2h), gRPC (https://grpc.io) и асинхронная передача событий. На момент написания книги использование форматов JSON, ProtoBuf или Avro для передачи сообщений в Kafka Streams, судя по всему, было самым популярным вариантом. Не имеет значения, какой стиль коммуникации вы выберете. Любой из них предполагает обмен сообщениями, и их формат должен быть хорошо задокументирован и являться частью «контракта» обмена. Для каждого из этих стилей можно применить методологию SEED(S) способом, подходящим для конкретного стиля. Поскольку RESTful API, вероятно, является самым простым и по-прежнему наиболее распространенным интерфейсом, мы продемонстрируем подход с использованием RESTful, но сама методология применима и к другим стилям.

Вы можете использовать любой инструмент для редактирования и создания своих спецификаций OAS. Однако если вы ищете вариант с открытым исходным кодом, доступный на большинстве платформ и хорошо себя зарекомендовавший, то таковым является редактор VS Code (https://code.visualstudio.com) с плагином Open API Designer (https://oreil.ly/LySwF). Установив плагин и открыв файл описания YAML, при активной вкладке нажмите CTRL+ALT+P (в Windows) или CMD+ALT+P (в macOS) и выберите соответствующую команду предварительного просмотра, чтобы увидеть отображение спецификации, как показано на рис. 3.3.

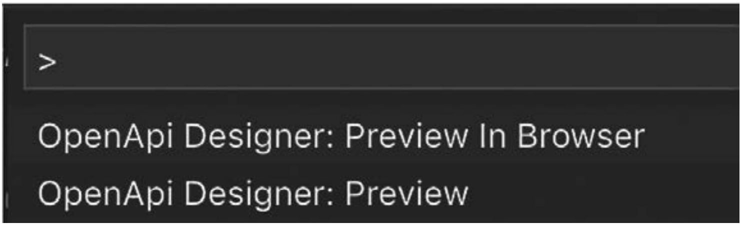


Рис. 3.3. Выбор предварительного просмотра OAS в VS Code

Пример OAS для действия в нашем проекте

Простая версия OAS для действия перебронирования, описанного выше в этой главе, может выглядеть примерно следующим образом:

```
openapi: 3.0.0
info:
  title: Airline Reservations Management API
```

```
  description: |
    API for Airline Management System
  version: 1.0.1
servers:
  - url: http://api.example.com/v1
    description: Production Server
paths:
  /reservations/{reservation_id}:
    put:
      # @см. https://swagger.io/docs/specification/describing-parameters
      summary: Book or re-book a reservation
      description: |
        Example request:
        ```
          PUT http://api.example.com/v1/reservations/d2783fc5-0fee
        ```
      parameters:
        - name: reservation_id
          in: path
          required: true
          description: Unique identifier of the reservation being created or
                       changed
          schema:
            type : string
          example: d2783fc5-0fee

      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                outbound:
                  type: object
                  properties:
                    flight_num:
                      type: string
                      example: "AA 253"
                    flight_date:
                      type: string
                      example: "2019-12-31T08:01:00"
                    seats:
                      type: array
                      items:
                        type: string
                returning:
                  type: object
                  properties:
                    flight_num:
```

```
          type: string
          example: "AA 254"
        flight_date:
          type: string
          example: "2020-01-07T14:16:00"
        seats:
          type: array
          items:
            type: string
            example: [
              {
                outbound: {
                  flight_num: "AA 253",
                  flight_date: "2019-12-31T08:01:00",
                  seats: [
                    "9C"
                  ]
                },
                returning: {
                  flight_num: "AA 254",
                  flight_date: "2020-01-07T14:16:00",
                  seats: [
                    "10A"
                  ]
                }
              }
            ]


      responses:
        '200': # успешный ответ
          description: Successful Reservation
          content:
            application/json:
              schema:
                type: object
                properties:
                  reservation_id:
                    type: string
                    description: some additional description
        '403':
          description: seat(s) unavailable. Booking failed.
          content:
            application/json:
              schema:
                type: string
                description: detailed information
```

Представление, полученное с помощью плагина VS Code, должно выглядеть примерно так, как показано на рис. 3.4.

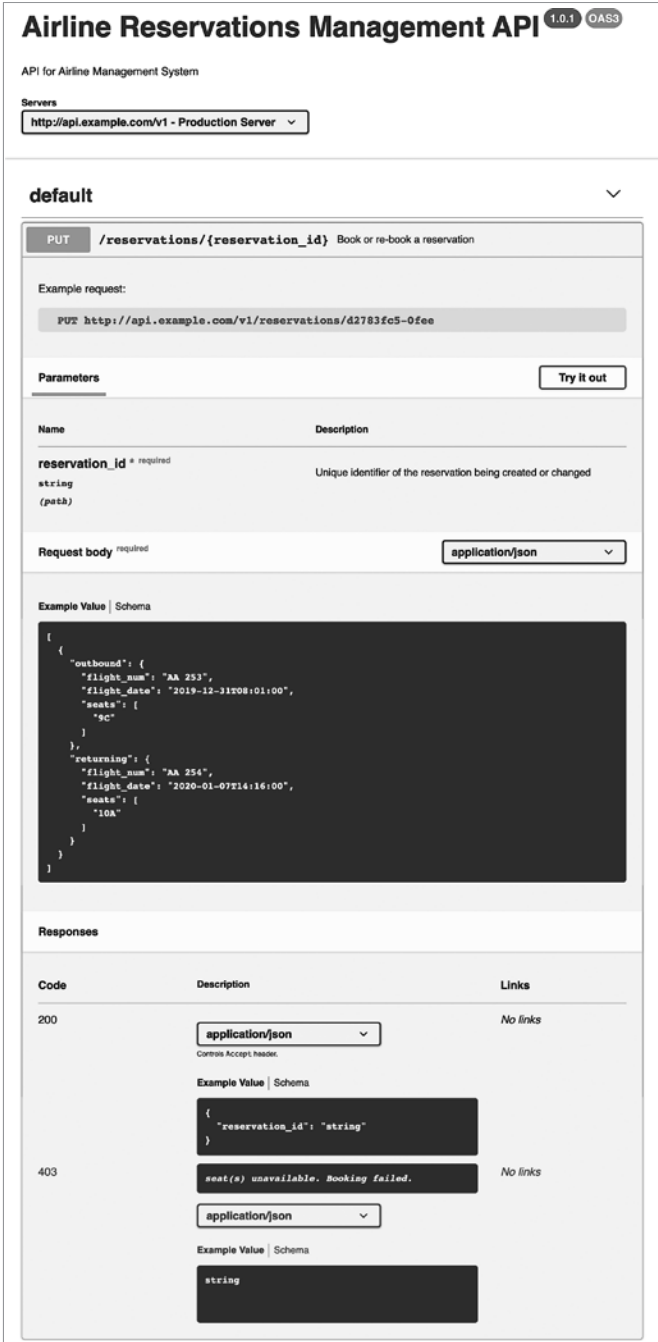


Рис. 3.4. Визуализация образца документа OAS

Формальное определение контракта API — важная веха в разработке проекта API и микросервисов. Некоторые могут даже посчитать это отлично выполненной работой на данный момент. Однако проектирование API не заканчивается на текущем этапе. Мы хотели бы, чтобы все было так просто, но на самом деле есть еще одно важное мероприятие, которое необходимо завершить. Следующий шаг в процессе SEED(S) фиксирует это действие.

Получение обратной связи по спецификации API

Начальная версия проекта API и сервиса, отраженная в описании, основанном на OAS или в каком-либо другом стандарте, является важной вехой, но для хорошо спроектированного API этого недостаточно.

Нам нужно показать эскизный проект конечных точек разработчикам клиентов, которым будет предложено использовать эти API и сервисы, и получить их отзывы. Если предыдущие этапы включали активный мозговой штурм и работу, то данный этап состоит во внимательном выслушивании и размышлении. Это невероятно важный шаг в разработке API для тех, кто желает создавать такие API и микросервисы, которые выдержат испытание временем и понравятся вашим клиентам.

> **КЛЮЧЕВОЕ РЕШЕНИЕ: ПОЛУЧИТЬ ОТЗЫВЫ О ПРОЕКТАХ СЕРВИСОВ**
>
> Проектирование сервиса не завершается, пока его проект не будет представлен целевой аудитории, а полученные отзывы не будут учтены.

Как правило, при разработке сервисов и API необходимо иметь в виду две группы клиентов:

- *конечные пользователи системы* — ваши API обеспечивают пользовательский опыт;
- *разработчики клиентов, которые будут программировать взаимодействия с вашими сервисами (API или микросервисами)*, — они формируют пользовательский опыт, например, в веб- или мобильных приложениях.

В начале процесса SEED(S) мы проводим собеседование с конечными пользователями, чтобы собрать и проанализировать истории заданий. Однако позже в процессе мы начинаем получать обратную связь от разработчиков клиентов. Это может произойти уже на этапе проектирования взаимодействий, а затем

снова после создания OAS, перед программированием. Эту вторую группу, разработчиков клиентов API, обязательно нужно опросить, чтобы оценить удобство использования проектируемых API и избежать создания чего-то, что может быть отвергнуто ими из-за неудобства использования.

Оба вида исследовательской деятельности имеют решающее значение. Первый гарантирует создание правильной системы, а последний — удобство ее использования!

Реализация микросервисов

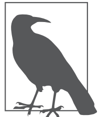
Последний шаг в методологии SEED(S) — фактическая реализация микросервисов. Этот этап намеренно поставлен в самый конец процесса. Программирование — один из самых дорогостоящих видов деятельности в любой команде разработчиков программного обеспечения. Перепрограммирование функциональности, изначально разработанной на основе неправильных предположений, — жутко трудоемкая и дорогостоящая задача. Вот почему мы следуем тщательно продуманному процессу, такому как SEED(S), прежде чем переходить к реализации микросервисов. В целом это помогает сэкономить время и добиться лучших результатов.

Прежде чем завершить эту главу, необходимо прояснить важную деталь. На протяжении всей главы мы говорили «API и микросервисы» и начали с упоминания о том, что методологию SEED(S) можно одинаково успешно применять как к процессу проектирования API, так и к процессу разработки микросервисов. Отчасти это верно, поскольку API и микросервисы имеют много общего. Но чем они отличаются друг от друга? Являются ли микросервисы просто небольшими API? В следующем разделе мы попытаемся ответить на этот важный вопрос.

Микросервисы и API

API и микросервисы действительно имеют много общего. Микросервисы — это возможности, предоставляемые через стандартные сетевые протоколы, чаще всего HTTP. Но возможности, предоставляемые в виде конечных точек HTTP, были известны как веб-API задолго до появления термина «микросервисы». Итак, являются ли эти два понятия по сути одним и тем же? Являются ли микросервисы просто новым видом API — уменьшенными API? Что еще более важно, нужны ли нам вообще обычные API, когда мы начинаем писать микросервисы, или уменьшенные API (микросервисы) заменяют большие (обычные) API? Мы часто видели, как эти вопросы вызывают путаницу в командах, пытающихся внедрить архитектуру микросервисов.

Периодически мы сталкивались с разработчиками, называющими «микросервисами» любые небольшие API, сфокусированные на определенной задаче. При таком подходе микросервисы играют ту же роль, что и предшествовавшие им API, поэтому они действительно заменяют старые API. По нашему опыту, такой способ представления микросервисов неидеален, и мы предлагаем альтернативное, хотя и самоуверенное, определение отличий микросервисов от устаревших API. Наш подход основан на опыте некоторых известных экспертов в этой области и на собственном опыте работы с успешными проектами и командами микросервисов.



Микросервисы — это не просто уменьшенные API

Микросервисы — это не просто уменьшенные замены прежних API. Микросервисы обеспечивают реализацию вашей системы, в то время как API попрежнему играют роль внешнего интерфейса системы.

Мы считаем, что если микросервисы что-то заменяют, то они заменяют модульные компоненты, которые вы использовали для создания своих систем. Если раньше вы собирали большую систему, связывая (статически или динамически) различные модули, то в архитектуре микросервисов строительными блоками являются сетевые сервисы, которые мы называем «микросервисами». Этот подход показан на рис. 3.5.

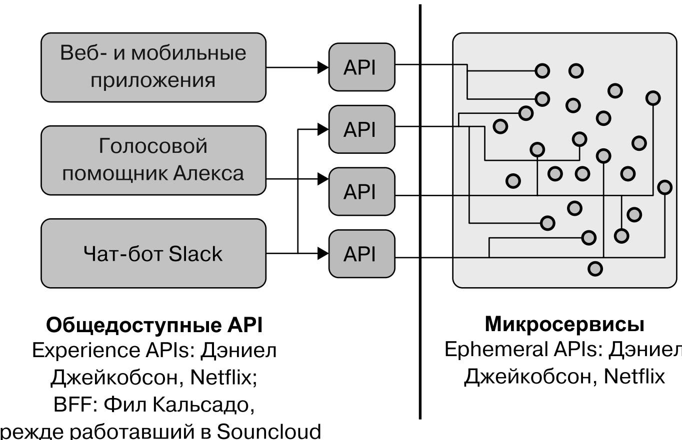


Рис. 3.5. Взаимосвязь между микросервисами и API

Обратите внимание, что аналогичный подход — разделение API на «внутренние, с которыми вы создаете свои сервисы», и «внешние, которые оптимизированы для нужд пользовательских интерфейсов» — описали Фил Кальсадо как подход

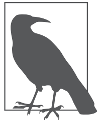
«бэкенд для фронтенда» (backend for frontend, BFF) (https://oreil.ly/ef8jV), когда работал в SoundCloud, и Дэниел Джейкобсон (https://oreil.ly/CxTka) во время своей работы в Netflix. Дэниел Джейкобсон объяснил, как в Netflix они разделили API на Experience (фронтенд — внешний) и Ephemeral (бэкенд — внутренний).

КЛЮЧЕВОЕ РЕШЕНИЕ: ВЕБ-API РАСПОЛОЖЕНЫ ПОВЕРХ МИКРОСЕРВИСОВ

Необходимо различать веб-API, представляющие общедоступный интерфейс вашей системы, и микросервисы, представляющие реализацию той же системы. Старайтесь не думать о микросервисах как о «просто небольших API».

Не существует единственно правильного способа организации микросервисов и подключения их к «внешнему» API. И сейчас мы собираемся выполнить обещание представить в этой книге беззастенчиво самоуверенные рекомендации. Наши мнения основаны на успешных примерах, но мы также признаем, что другие специалисты могли бы с не меньшим успехом применить другие стратегии.

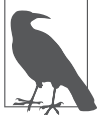
По нашему опыту, идеальное разделение обязанностей происходит, когда вся бизнес-логика (возможности) реализуется микросервисами, а API действуют как тонкий слой оркестрации перед этими микросервисами. Кроме того, мы рекомендуем командам стараться избегать прямых взаимодействий между микросервисами. Вместо этого ради ослабления связи лучше реализовать процесс оркестрации на уровне API перед микросервисами, чтобы микросервисы ничего не знали друг о друге.



Обратите внимание, что между API и микросервисами, реализующими соответствующие возможности, нет прямой связи. Эти два актива — части принципиально разных уровней вашей архитектуры.

Мы считаем, что «микросервисы не должны ничего знать друг о друге и управляться извне» — это подход, в котором философия Unix (https://oreil.ly/GVjJV) построения системы как набора составных инструментов хорошо согласуется с принципами архитектуры микросервисов. Один из наиболее важных аспектов философии Unix — возможность комбинировать инструменты Unix (например, инструменты GNU) различными способами, используя конвейер ввода/вывода в командной строке или в сценариях оболочки. Однако для достижения этой цели крайне важно, чтобы различные инструменты Unix действовали одинаково для любого ввода — им должно быть все равно, кто их «вызывает» или куда отправляются их выходные данные. Компоненты не обязаны знать хоть что-то друг о друге, чтобы использоваться в комбинации. Слабая связанность заставляет всю систему работать, а не только означает наличие маленьких

и сфокусированных на определенных задачах инструментов. То же относится и к микросервисам.



Держите микросервисы в неведении друг о друге

Избегайте просачивания в микросервисы «знаний» друг о друге и прямых взаимодействий через синхронные интерфейсы. Вместо этого попробуйте организовать процессы, использующие несколько микросервисов на уровне API. Если такое решение недоступно, подумайте о применении асинхронных интерфейсов, когда вышестоящий микросервис публикует данные в журнале событий (например, Kafka), а нижестоящий может подписаться на этот журнал событий, чтобы исключить создание тесной связи между вышестоящим микросервисом и подписчиком (подписчиками).

Резюме

В этой главе мы заложили важнейшую основу для понимания процесса проектирования надежных микросервисов. Взяв на вооружение эффективную и повторяемую методологию, SEED(S), мы определили, какие черты делают проекты успешными на пути к микросервисам, и научились адаптировать эти черты к собственным обстоятельствам.

В следующих главах мы будем использовать идеи, полученные в ходе знакомства с SEED(S). В главах 4 и 5 мы углубимся в процесс проектирования микросервисов, а в главе 9 рассмотрим код, реализующий несколько микросервисов нашего примера проекта.

ГЛАВА 4

Выбор оптимального размера микросервисов: определение границ сервисов

Один из наиболее сложных аспектов построения успешной системы микросервисов — правильное определение их границ. Интуитивно понятно, что разбиение большой кодовой базы на более мелкие, простые и слабосвязанные части упрощает сопровождение. Но как определить, где и как разделить код на части, чтобы достичь желаемых свойств? Какие правила использовать, чтобы понять, где заканчивается один сервис и начинается другой? Ответить на эти фундаментальные вопросы непросто. Многие команды, новички в микросервисах, сталкиваются с ними. Неправильное определение границ микросервиса может значительно уменьшить преимущества его использования, а в некоторых случаях даже свести на нет все усилия. Поэтому неудивительно, что наиболее частый и насущный вопрос, который задают специалисты по микросервисам, звучит следующим образом: как правильно разделить крупное приложение на набор микросервисов?

В этой главе мы подробно рассмотрим методологию эффективного анализа, моделирования и декомпозиции больших предметных областей (предметно-ориентированное проектирование — Domain-Driven Design), объясним преимущества использования метода Event Storming для анализа предметной области и завершим представлением универсальной формулы определения размера, уникального руководства по эффективному определению размера микросервисов.

Почему границы имеют значение, когда они имеют значение и как их найти

В самом названии архитектурного шаблона присутствует слово *«микро»*: архитектура, которую мы разрабатываем, — это архитектура микросервисов! Но насколько «микро» должны быть наши сервисы? Очевидно, что речь не идет о физической длине чего-либо и не предполагается, что *микро* означает одну миллионную долю метра (то есть базовой единицы длины в Международной системе единиц). Так что же означает *микро* для нас? Как разделить нашу большую задачу на мелкие сервисы, чтобы получить обещанные преимущества микросервисов? Может быть, нужно распечатать исходный код на бумаге, склеить все листы вместе и измерить его буквальную длину? Или, если без шуток, следует руководствоваться количеством строк в исходном коде — и постараться сохранить это число небольшим, чтобы гарантировать, что каждый из микросервисов также получится достаточно малым? Однако что значит «достаточно»? Можно ли просто произвольно объявить, что каждый микросервис должен содержать не более 500 строк кода? А может быть, можно провести границы по сторонам функциональных блоков и сказать, что каждая отдельная возможность, представленная функцией в исходном коде нашей системы, является микросервисом? Следуя этой логике, мы могли бы реализовать все наше приложение, скажем, с помощью бессерверных функций, объявив каждую такую функцию микросервисом. Легко и просто! Верно? А может быть, и нет.

На самом деле все эти упрощенные подходы были опробованы на практике, и все они имеют существенные недостатки. Традиционно исходные строки кода (source lines of code, SLOC) использовались в качестве меры затрат усилий/сложности, но ныне общепризнанно, что это плохая мера определения сложности или истинного размера любого кода, которым легко манипулировать. Поэтому, даже если бы нашей целью было создание мини-сервисов в надежде сохранить их простыми, количество строк кода было бы плохим показателем.

Проведение границ по краям функциональных блоков еще более заманчиво. И стало еще более соблазнительным с ростом популярности бессерверных функций, таких как лямбда-функции Amazon Web Services (AWS). Опираясь на производительность и широкое внедрение лямбд AWS, многие команды поспешили объявить эти функции микросервисами. Если вы пойдете по этому пути, то возникнет ряд проблем, наиболее серьезными из которых являются следующие.

- *Проведение границ на основе технических потребностей является антипаттерном.* Согласно Льюису и Фаулеру (https://oreil.ly/mRUrv), микросервисы должны быть «организованы вокруг бизнес-возможностей», а не технических потребностей. Аналогично Парнас (https://oreil.ly/1AcI0) в статье 1972 года

рекомендует разделять системы, основываясь на модульной инкапсуляции изменений проекта с течением времени. Ни один из подходов не имеет строгого соответствия границам бессерверных функций.

- *Слишком подробная детализация, слишком рано.* Взрывная детализация на ранних этапах проектирования микросервисов может привести к чрезмерному усложнению, способному затормозить работу над микросервисами еще до того, как у них появится шанс добиться успеха.

В главе 1 мы сформулировали основную цель архитектуры микросервисов: в первую очередь минимизировать затраты на координацию в сложной среде с несколькими командами для достижения гармонии между скоростью и безопасностью. Поэтому сервисы должны проектироваться так, чтобы свести к минимуму необходимость координации между командами, работающими над различными микросервисами. Однако если разделить код на функции так, что это не обязательно приведет к минимизации координации, то мы получим микросервисы неправильного размера. Просто предполагать, что любой способ организации кода в бессерверные функции уменьшит координацию, ошибочно.

Ранее мы заявляли, что важной причиной отказа от подхода к разделению приложения на микросервисы, основанного на размере или функциональных границах, является опасность преждевременной оптимизации — слишком большого количества слишком маленьких сервисов на ранних этапах вашего пути к микросервисам. Первопроходцы в мире микросервисов, такие как Netflix, SoundCloud, Amazon и другие, в конечном итоге обнаружили, что у них слишком много микросервисов (https://oreil.ly/r5vYU)! Однако это не означает, что они начали с сотен узкоспециализированных микросервисов в первый же день. Скорее, большое количество микросервисов — это то, до чего они оптимизировали свои системы за многие годы разработки, *после* достижения операционной зрелости, позволяющей справиться с уровнем сложности, обусловленным высокой степенью детализации микросервисов.

Избегайте создания избыточного количества микросервисов на ранних этапах

Определение размера сервисов в микросервисной архитектуре, безусловно, является процессом, который должен разворачиваться со временем. Верный способ саботировать все усилия — попытаться разработать чрезмерно детализированную систему на ранних этапах этого процесса.

Независимо от того, работаете ли вы над новым проектом или разделяете на части существующий монолит, начинать нужно с небольшого количества сервисов и постепенно увеличивать его. Если это приведет к тому, что некоторые из микросервисов изначально получатся больше, чем в их целевом состоянии, то это совершенно нормально. Вы сможете разделить их позже.

Даже если мы начинаем с нескольких микросервисов и делаем это постепенно, нам нужна надежная методология определения их размера. Далее мы рассмотрим практические рекомендации, успешно применяемые в отрасли.

Предметно-ориентированное проектирование и границы микросервисов

В начале изучения успешных приемов проектирования микросервисов Сэм Ньюман представил некоторые основополагающие базовые правила в своей книге *Building Microservices*[1] (O'Reilly) (http://shop.oreilly.com/product/0636920033158.do). Он предположил, что при определении границ сервисов мы должны стремиться к такому проекту, чтобы сервисы были:

- *слабосвязанными* — сервисы должны быть достаточно неосведомленными и независимыми друг от друга, чтобы изменение кода в одном из них не приводило к возникновению волнового эффекта в других. Желательно также ограничить количество различных типов вызовов от одного сервиса к другому, поскольку, помимо потенциальной проблемы с производительностью, интенсивное общение также может привести к тесной связанности компонентов. Учитывая наш подход «минимизации координации», преимущество слабой связи сервисов совершенно очевидно;
- *высокосвязными* — функции, присутствующие в сервисе, должны быть тесно связаны, в то время как несвязанные функции должны инкапсулироваться в другом месте. Таким образом, если требуется изменить логическую функциональную единицу, вы сможете изменить ее в одном месте и минимизировать время на публикацию этого изменения (важный показатель). Напротив, если для этого придется изменить код в нескольких сервисах, то понадобится выпустить множество различных сервисов одновременно, чтобы внедрить это изменение. Это потребует значительных затрат на координацию, особенно если этими сервисами «владеют» несколько разных команд, и напрямую поставит под угрозу нашу цель минимизации затрат на координацию;
- *согласованными с бизнес-возможностями* — большинство запросов на модификацию или расширение функциональности обусловлены потребностями бизнеса, и, если наши границы хорошо согласованы с границами бизнес-возможностей, то нам будет проще удовлетворить указанные выше первое и второе требования к проекту. Во времена монолитных архитектур программисты часто пытались стандартизировать «канонические модели данных». Однако

[1] *Ньюмен С.* Создание микросервисов. — СПб.: Питер, 2023.

> практика вновь и вновь демонстрировала, что подробные модели данных, моделирующие реальность, существуют недолго — они довольно часто меняются и их стандартизация приводит к частым переделкам. Набор бизнес-возможностей, предоставляемых вашими подсистемами, напротив, является более долговечным. Модуль учета всегда сможет предоставить желаемый набор возможностей более крупной системе независимо от того, как ее внутренняя организация будет развиваться с течением времени.

Эти принципы проектирования доказали свою полезность и получили широкое распространение среди разработчиков микросервисов. Тем не менее они являют собой довольно высокоуровневые амбициозные принципы и, возможно, не предоставляют конкретных рекомендаций по выбору сервисов, необходимых практикам в их повседневной работе. В поисках более практичной методологии многие обратились к предметно-ориентированному проектированию.

Методология проектирования программного обеспечения, известная как предметно-ориентированное проектирование (Domain-Driven Design, DDD), появилась значительно раньше архитектуры микросервисов. Ее представил Эрик Эванс в 2003 году в своей одноименной книге *Domain-Driven Design: Tackling Complexity in the Heart of Software*[1] (Addison-Wesley) (https://learning.oreilly.com/library/view/domain-driven-design-tackling/0321125215). Основной предпосылкой появления методологии является утверждение о том, что при анализе сложных систем мы должны избегать поиска единой универсальной модели предметной области, представляющей всю систему. В частности, как сказал Эванс в своей книге,

> «в больших проектах сосуществует несколько моделей, и во многих случаях это прекрасно работает. Разные модели применяются в разных контекстах».

Как только Эванс установил, что сложная система, по сути, представляет собой набор моделей нескольких предметных областей, он сделал важный дополнительный шаг, введя понятие *ограниченного контекста*. В частности, он заявил, что:

> «ограниченный контекст определяет диапазон применимости каждой модели».

Ограниченные контексты позволяют реализовывать и выполнять различные части более крупной системы, не искажая присутствующих в ней независимых моделей предметной области. После выявления ограниченных контекстов Эрик также любезно предоставил формулу определения оптимальных границ такого контекста, установив концепцию *единого языка* (ubiquitous language).

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.

Чтобы понять значение единого языка, важно отметить следующее. Четко определенная модель предметной области в первую очередь обеспечивает общий словарь определенных терминов и понятий, общий язык для описания предметной области, который эксперты и инженеры разрабатывают совместно в тесном сотрудничестве, согласуя бизнес-требования и реализацию. Этот общий язык, или общий словарный запас, — то, что в DDD мы называем единым языком. Важность данного наблюдения заключается в признании, что одни и те же слова могут иметь разное значение в разных ограниченных контекстах. Классический пример этого показан на рис. 4.1. Термин *account* имеет существенно иное значение в контексте управления идентификацией и доступом, управления клиентами и финансового учета в системе онлайн-бронирования.

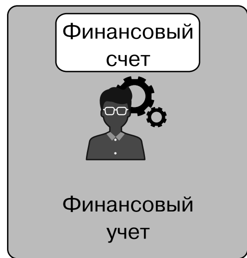


Рис. 4.1. Значения слова account различаются в разных предметных областях

Действительно, в контексте управления идентификацией и доступом учетная запись (account) — это набор учетных данных, используемых для аутентификации и авторизации. В контексте управления клиентами аккаунт — это набор демографических и контактных данных, в то время как в контексте финансового учета — это платежная информация и список прежних транзакций. Мы можем видеть, что одно и то же базовое английское слово используется с принципиально разным значением в разных контекстах, и это нормально, поскольку нам нужно только согласовать единое значение терминов (единый язык) в ограниченном контексте конкретной модели предметной области. Согласно DDD, наблюдая границы, по которым термины меняют свое значение, можно определить границы контекстов.

В DDD не все термины, приходящие на ум при обсуждении модели предметной области, попадают в соответствующий единый язык. Концепции в ограниченном контексте, которые являются ключевыми для основной цели контекста, выступают частью единого языка команды; все остальные должны быть опущены. Эти основные понятия можно узнать из набора JTBD, который вы создаете для ограниченного контекста. В качестве примера посмотрим на рис. 4.2.

История задания
Пользователь регистрируется.
Когда клиент пытается
зарегистрироваться в системе,
ему хотелось бы ввести
имя пользователя,
и ввести пароль,
и ввести однократный пин-код,
чтобы получить данные,
аутентифицированные системой
и учетными данными

Рис. 4.2. Использование синтаксиса истории заданий для определения ключевых терминов единого языка

В этом примере мы используем формат истории заданий, представленный в главе 3, и применяем его к заданию из ограниченного контекста идентификации и управления доступом. Мы видим, что ключевые слова, выделенные на рис. 4.2, соответствуют терминам в едином языке. Мы настоятельно рекомендуем применять технику использования ключевых слов из историй заданий при определении словарных терминов, имеющих отношение к вашему единому языку.

Теперь, обсудив некоторые ключевые концепции DDD, перейдем к следующему разделу и рассмотрим кое-что, что может пригодиться при проектировании взаимодействий между микросервисами: составление карты контекста.

Составление карты контекста

В DDD мы не пытаемся описать сложную систему с помощью одной модели предметной области. Скорее разрабатываем несколько независимых моделей, сосуществующих в системе. Эти подобласти обычно взаимодействуют, используя опубликованные описания интерфейса. Представление различных областей в более крупной системе и способов их взаимодействий называется *картой контекста*. Отсюда акт идентификации и описания указанных взаимодействий известен как *составление карты контекста* (рис. 4.3).

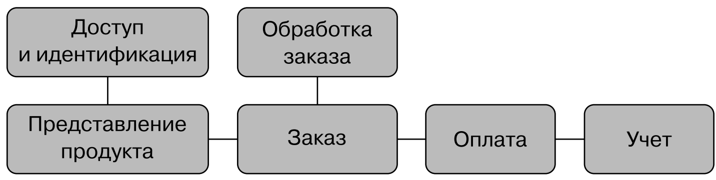


Рис. 4.3. Составление карты контекста

DDD определяет несколько основных типов взаимодействий в совместной работе при составлении карт ограниченных контекстов. Самый базовый тип известен как *общее ядро* (shared kernel). Оно возникает, когда две области разрабатываются практически независимо и в конце почти случайно перекрываются на некотором подмножестве точек друг друга (рис. 4.4). Две стороны могут договориться о совместной работе над этим общим ядром, которое может включать общий код и модель данных, а также описание предметной области.

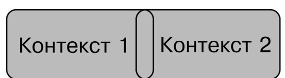


Рис. 4.4. Общее ядро

Несмотря на то что на первый взгляд все выглядит заманчиво (в конце концов, стремление к сотрудничеству — один из человеческих инстинктов), общее ядро представляет собой проблемный шаблон, особенно в микросервисных архитектурах. По определению, общее ядро требует высокой степени координации между двумя независимыми командами при создании и продолжает требовать координации согласований для любых дальнейших модификаций. Добавление общих ядер в микросервисную архитектуру приводит к появлению множества точек координации. В тех случаях, когда общее ядро действительно необходимо в экосистеме микросервисов, рекомендуется назначить одну команду основным владельцем/куратором, а все остальные — участниками.

В качестве альтернативы два ограниченных контекста могут вступать в отношения, которые в DDD называются отношениями типа «вышестоящий — нижестоящий» (Upstream — Downstream). В этом типе отношений вышестоящий контекст действует как поставщик некоторой возможности, а нижестоящий — как потребитель. Поскольку определения предметной области и реализации не пересекаются, этот тип отношений дает более слабую связанность, чем общее ядро (рис. 4.5).

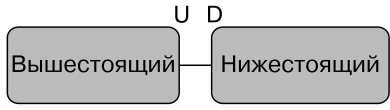


Рис. 4.5. Отношение «вышестоящий — нижестоящий»

В зависимости от типа координации и связи, отношение «вышестоящий — нижестоящий» можно представить в нескольких формах.

- *Клиент — поставщик.* В этом сценарии вышестоящий контекст (поставщик) предоставляет функциональность нижестоящему (клиенту). Пока предоставляемая функциональность ценна, все довольны. Однако вышестоящий контекст несет накладные расходы, связанные с обратной совместимостью. При изменении вышестоящего сервиса (поставщика) необходимо убедиться, что эти изменения не нарушат работу клиента. Что еще более важно, нижестоящий контекст (клиент) рискует своей работоспособностью из-за того, что поставщик намеренно или непреднамеренно что-то повредит или проигнорирует будущие потребности клиента.
- *Конформист.* Такие отношения — крайний случай рисков для взаимодействия клиента и поставщика. Это разновидность сценария «вышестоящий — нижестоящий», когда поставщик явно не заботится о потребностях своего клиента или не может этого делать. Это взаимодействие типа «на свой страх и риск». Поставщик предоставляет некоторые ценные возможности, в которых заинтересован клиент, но, учитывая, что поставщик не будет удовлетворять его потребности, клиент должен постоянно подстраиваться под изменения в сервисе-поставщике.

Конформистские отношения часто возникают в крупных организациях и системах, когда гораздо более крупная подсистема используется более мелкой. Представьте, что вы разрабатываете небольшую новую функцию внутри системы бронирования авиабилетов и вам нужно использовать, скажем, корпоративную платежную систему. Такая крупная корпоративная система вряд ли уделит время какой-то небольшой новой инициативе, но и вы не можете реализовать всю платежную систему самостоятельно. Поэтому вам придется либо стать конформистом, либо использовать другое жизнеспособное решение, прибегнув к *разделению путей*. Последнее не всегда означает реализацию аналогичной функциональности. Платежные системы слишком сложны, и ни одной маленькой команде не под силу реализовать такую систему как побочную задачу для достижения другой цели. Но вы можете выйти за пределы своего предприятия и использовать услуги коммерческой платежной системы, если ваша компания это позволяет.

В дополнение к конформизму и разделению путей у нижестоящего контекста есть еще несколько санкционированных DDD способов защитить себя от небрежности сервиса-поставщика: слой предотвращения повреждений и использование поставщика, предлагающего интерфейсы с открытым протоколом.

- *Слой предотвращения повреждений.* В данном сценарии клиент создает слой преобразования, называемый *антикоррозийным* или слоем предотвращения повреждений (anticorruption layer, ACL), между едиными языками, своим и поставщика, чтобы защитить себя от будущих критических изменений в интерфейсе поставщика. Создание ACL — эффективная и иногда необходимая

мера защиты, но команды должны иметь в виду, что в долгосрочной перспективе она может быть довольно дорогостоящей для сервисов-клиентов (рис. 4.6).

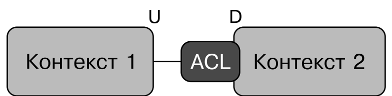


Рис. 4.6. Слой предотвращения повреждений

- *Сервис с открытым протоколом* (open host service, OHS). Допустим, поставщик знает, что его возможности могут использовать несколько разных клиентов. Тогда вместо того, чтобы пытаться координировать потребности своих многочисленных текущих и будущих потребителей, он может определить и опубликовать стандартный интерфейс, который все потребители должны будут принять. В DDD такие поставщики известны как сервисы с открытым протоколом. Предлагая открытый простой протокол для интеграции со всеми авторизованными сторонами и поддерживая обратную совместимость указанного протокола или обеспечивая четкое и безопасное управление его версиями, сервис с открытым протоколом может масштабировать свои операции без особых проблем. Практически все общедоступные сервисы (API) используют этот подход. Например, когда вы используете API поставщика общедоступных облачных сервисов (AWS, Google, Azure и т. д.), они обычно не знают и не обслуживают вас конкретно, поскольку у них миллионы клиентов, но могут предоставлять и развивать полезную услугу, работая как сервисы с открытым протоколом (рис. 4.7).

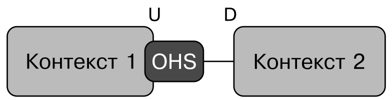


Рис. 4.7. Сервис с открытым протоколом

В дополнение к типам отношений между предметными областями карты контекстов также могут различаться типами интеграции между ограниченными контекстами.

Синхронные и асинхронные интеграции

Интерфейсы интеграции между ограниченными контекстами могут быть синхронными или асинхронными, как показано на рис. 4.8. Ни одна из моделей интеграции принципиально не предполагает того или иного стиля.

Типичными представителями модели синхронной интеграции между контекстами являются RESTful API, доступные через HTTP, сервисы gRPC, использующие двоичные форматы, такие как protobuf, и более новые сервисы, использующие интерфейсы GraphQL.

На асинхронной стороне лидируют взаимодействия типа «публикация — подписка». В этой модели взаимодействия поставщики могут генерировать некоторые события, а клиенты используют исполнителей (обработчиков) для обработки интересующих их событий, как показано на рис. 4.8.

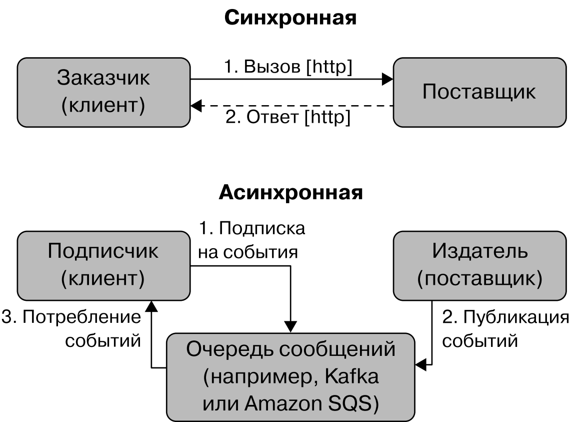


Рис. 4.8. Синхронные и асинхронные интеграции

Взаимодействия вида «публикация — подписка» сложнее в реализации и отладке, но они могут обеспечить превосходный уровень масштабируемости, устойчивости и гибкости, поскольку несколько получателей, даже реализованных с применением разных стеков технологий, могут подписываться на одни и те же события, используя единый подход и реализацию.

Чтобы завершить обсуждение ключевых концепций предметно-ориентированного проектирования, мы должны изучить концепцию агрегатов. Обсудим ее в следующем подразделе.

Агрегаты в DDD

В DDD *агрегат* — это набор связанных предметных объектов, которые внешние потребители могут рассматривать как единое целое. Эти внешние потребители ссылаются только на одну сущность в агрегате, известную в DDD как *корень агрегата* (aggregate root). Агрегаты позволяют предметным областям скрывать внутреннее устройство и предоставлять только ту информацию и возможности

(интерфейс), которые «интересны» внешнему потребителю. Например, в отношениях «поставщик — клиент», обсуждавшихся выше, клиент не должен и, как правило, не захочет знать о каждом отдельном предметном объекте в поставщике. Вместо этого он будет рассматривать поставщика как агрегат или совокупность агрегатов.

Мы снова встретимся с понятием агрегата в следующем разделе, когда будем обсуждать Event Storming — эффективную методологию, значительно упрощающую процесс анализа предметной области и превращающую его в гораздо более быстрое и увлекательное упражнение.

Введение в Event Storming

Предметно-ориентированное проектирование — эффективная методология для анализа как общесистемного (называемого «стратегическим» в DDD), так и углубленного (называемого «тактическим») уровней больших и сложных систем. Мы также видели, что анализ DDD может помочь идентифицировать автономные подкомпоненты, слабосвязанные через ограниченные контексты соответствующих предметных областей.

Очень легко прийти к следующему выводу: чтобы научиться правильно определять размеры микросервисов, достаточно хорошо разбираться в анализе предметной области. Если мы заставим всю нашу компанию изучить и полюбить эту методологию (поскольку DDD, безусловно, командный вид спорта), то мы окажемся на пути к успеху!

На заре микросервисных архитектур DDD было настолько широко провозглашено *единственным верным способом* определения размера микросервисов, что их развитие также дало огромный толчок практике DDD, по крайней мере больше людей узнали об этом подходе к проектированию и ссылались на него. Внезапно многие докладчики заговорили о DDD на всевозможных конференциях по программному обеспечению и многие команды начали утверждать, что используют этот способ проектирования в повседневной работе. Увы, при ближайшем рассмотрении легко обнаружить, что реальность была несколько иной и DDD стало одним из подходов, о которых «много говорят, но мало практикуют».

Не поймите нас неправильно: были люди, которые прибегали к DDD задолго до появления микросервисов, и многие используют его и сейчас, но, говоря конкретно о его применении в качестве инструмента определения размера микросервисов, это было больше шумихой и пародией, чем реальностью.

Есть две основные причины, почему люди больше говорили о DDD, чем практиковали его всерьез, — это сложно и дорого. Чтобы применять DDD, нужны

обширные знания и опыт. Оригинальная книга Эрика Эванса на эту тему насчитывает 520 страниц, и вам нужно прочитать как минимум еще несколько книг, чтобы действительно понять тему, не говоря уже о том, чтобы получить некоторый опыт практического применения этого способа на практике. Просто не хватало людей с необходимыми навыками и опытом, а кривая обучения была крутой.

Хуже того, как мы уже упоминали, DDD — командный вид спорта, к тому же требующий большого количества времени. Недостаточно иметь парочку технологов, хорошо разбирающихся в DDD. Вам также нужно убедить команды разработчиков продукта, проектирования и т. д. принять участие в длительных и насыщенных встречах для обсуждения предметной области, не говоря уже о том, чтобы объяснить им хотя бы основы того, чего вы пытаетесь достичь. Итак, по большому счету, стоит ли оно того? Скорее всего, да: особенно большим, рискованным и дорогим системам DDD может дать множество преимуществ. Однако если вы просто хотите быстро продвинуться в процессе разработки и изменить размер некоторых микросервисов и уже нажили свой политический капитал на работе, продавая всем новинку под названием «микросервисы», — желаем удачно убедить множество занятых людей дать вам достаточно времени, чтобы выбрать оптимальный размер для ваших сервисов! Этого просто не случится — слишком дорого и отнимет много времени.

А потом вдруг парень по имени Альберто Брандолини (https://oreil.ly/TiPOb), потративший десятилетия на поиск лучших способов совместной работы команд, нашел кратчайший путь! Он предложил увлекательный, легкий и недорогой процесс Event Storming, в значительной степени основанный на идеях DDD и способный помочь найти ограниченные контексты за считаные часы, а не недели или месяцы. Внедрение Event Storming стало прорывом и способствовало удешевлению применения DDD и, в частности, упрощению определения размера сервиса. Конечно, это не полная замена и она не дает *всех* преимуществ формального подхода DDD (в противном случае это было бы волшебно).

Но что касается выявления ограниченных контекстов с хорошим приближением — это действительно волшебно!

Event Storming — очень эффективный метод, помогающий идентифицировать ограниченные контексты домена упрощенным, увлекательным и эффективным способом, обычно намного быстрее, чем при использовании традиционной методологии DDD. Этот прагматичный подход снижает стоимость анализа DDD настолько, что делает его жизнеспособным в ситуациях, когда DDD было бы недоступно. Давайте посмотрим, как работает «магия» Event Storming.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ EVENT STORMING ВМЕСТО ОФИЦИАЛЬНОГО DDD

Используйте более легкий процесс анализа событий вместо формального подхода DDD, чтобы выявить основные агрегаты в предметной области и провести границы различных ограниченных контекстов, присутствующих в вашей системе.

Процесс Event Storming

Красота Event Storming заключается в его гениальной простоте. Для проведения сеанса Event Storming вам понадобится очень длинная стена (желательно в физическом пространстве и чем длиннее, тем лучше), много принадлежностей, в основном стикеров и фломастеров, и 4–5 часов времени, уделенного представительными членами вашей команды. Для успешного проведения Event Storming крайне важно, чтобы участники были не только инженерами — существенное значение имеет участие представителей производства и бизнеса. Сеансы Event Storming также можно проводить виртуально, с помощью цифровых инструментов для организации совместной работы, которые могут имитировать физический процесс, описанный здесь.

Процесс проведения сеансов Event Storming вживую начинается с покупки расходных материалов. Чтобы упростить этот процесс для вас, мы создали список покупок на Amazon (https://oreil.ly/T7Y0i), включающий все, что мы используем в ходе сеансов Event Storming (рис. 4.9), в том числе следующее.

- Большое количество стикеров разных цветов, в первую очередь оранжевого и синего, а также нескольких других цветов для представления объектов различных типов. Вам понадобится действительно много таких стикеров. (В обычных магазинах их никогда не было в достаточном количестве, поэтому вошло в привычку покупать онлайн.)
- Рулон 12 мм белого бумажного скотча.
- Длинный рулон бумаги (например, бумага для рисования IKEA Mala), который будет вешаться на стену с помощью бумажного скотча. Лучше сделать несколько «дорожек».
- Маркеры, как минимум по количеству участников сеанса. У каждого должен быть собственный маркер!
- Мы уже упоминали длинную ровную стену, к которой можно приклеить рулон бумаги?

Рис. 4.9. Расходные материалы, требующиеся для проведения Event Storming

Во время сеансов Event Storming очень ценно широкое участие, например, экспертов в предметной области, владельцев продуктов и проектировщиков взаимодействий. Сеансы достаточно короткие (всего несколько часов, а не дни или недели, обычно уходящие на анализ). Учитывая ценность результатов сеансов, ясность, которую они вносят для всех представленных групп, и время, экономящееся в долгосрочной перспективе, это разумная трата времени для всех участников. Сеанс Event Storming, проводимый только с инженерами-программистами, по большей части бесполезен, поскольку происходит в ограниченном мирке и не может привести к межпредметным обсуждениям, необходимым для достижения желаемых результатов.

Итак, у нас есть расходные материалы, большая комната с длинной открытой стеной, к которой скотчем приклеен рулон бумаги, и все необходимые люди. Ведущий просит всех взять по несколько оранжевых стикеров и свой маркер. Затем дается простое задание: записать на оранжевых стикерах ключевые события анализируемой области (по одному событию на стикере), выраженные глаголами в прошедшем времени, и разместить эти стикеры вдоль временной шкалы на бумаге, приклеенной к стене, чтобы создать «дорожку» времени, как показано на рис. 4.10.

Введена информация о клиенте

Введена информация о кредитной карте

Платеж обработан

Посадочный талон сгенерирован

Посадочный талон показан

Посадочный талон отправлен на e-mail

Рис. 4.10. Временна́я линия событий со стикерами

Участники не должны зацикливаться на точной последовательности событий, и на данном этапе от участников не требуется согласовывать между собой порядок событий. Единственное, что им предлагается, — чтобы каждый придумал как можно больше событий и поместил более ранние, по их мнению, левее, а более поздние — правее. В задачу участников не входит отсеивать дубликаты. По крайней мере пока. Этот этап обычно занимает от 30 минут до часа в зависимости от масштаба задачи и количества участников. Обычно, чтобы назвать мероприятие успешным, необходимо, чтобы было создано не менее 100 заметок о событиях.

На втором этапе сеанса группе предлагают взглянуть на заметки на стене и с помощью ведущего начать упорядочивать их по времени, попутно удаляя дубликаты. При наличии достаточного количества времени весьма желательно, чтобы участники начали создавать «сюжетную линию», расставляя события в порядке, создающем что-то вроде «пути потребителя». На данном этапе у команды могут возникнуть вопросы или противоречия, но мы должны не решить их немедленно, а просто зафиксировать с помощью стикеров разного цвета (обычно фиолетовых) как «спорные точки». На вопросы в «спорных точках» мы будем отвечать позднее. Этот этап также может занять от 30 до 60 минут.

На третьем этапе создается то, что в Event Storming известно как *обратное повествование*. По сути, мы проходим временную шкалу от конца к началу и определяем *команды* — действия, вызвавшие события. Для обозначения команд используются стикеры другого цвета (обычно синего). На данном этапе ваша раскадровка может выглядеть примерно так, как показано на рис. 4.11.

Имейте в виду, что многие команды будут прямо связаны с событиями. Это может выглядеть как излишнее формулирование одного и того же в прошлом времени и в настоящем. И действительно, первые две команды на рис. 4.11 выглядят именно так. Это часто сбивает с толку людей, незнакомых с Event Storming. Просто не обращайте на это внимания! Мы не выносим суждений во время сеанса Event Storming, и хотя одни команды могут прямо соответствовать событиям, другие могут не иметь такого соответствия. Например, команда «Отправить учетные данные для платежа» запускает множество событий. Просто фиксируйте то, что, по вашему мнению, происходит в реальной жизни, и не старайтесь сделать все «красивым» или «аккуратным». В реальном мире, который вы моделируете, тоже много беспорядка.

Рис. 4.11. Добавление команд во временную линию Event Storming

На следующем этапе мы признаем, что команды не создают события напрямую. Их получают специальные типы предметных объектов и генерируют события. В Event Storming эти объекты называются *агрегатами* (да, название навеяно аналогичным понятием в DDD). На этом этапе мы перестраиваем наши команды и события, при необходимости нарушая упорядоченность во времени, чтобы сгруппировать команды вокруг агрегатов, в которые они отправляются, а события, «запущенные» этим агрегатом, поместить в него. Пример этого этапа Event Storming показан на рис. 4.12.

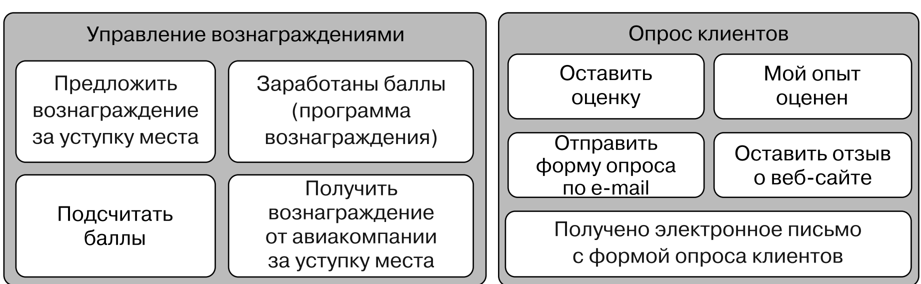


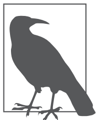
Рис. 4.12. Агрегаты на временной линии Event Storming

Этот этап сеанса может занять от 15 до 25 минут. По его завершении вы обнаружите, что стена теперь больше похожа на группы событий и команд вокруг агрегатов, а не на временну́ю шкалу событий. И представляете, эти кластеры являются ограниченными контекстами, которые мы искали.

Единственное, что осталось, — классифицировать различные контексты по уровню их приоритета (по аналогии с «корневыми», «вспомогательными» и «общими»

контекстами в DDD). Для этого мы создаем матрицу ограниченых контекстов/подобластей и упорядочиваем их по двум свойствам: сложности и конкурентному преимуществу. В каждой категории мы используем размеры футболок (S, M или L) для упорядочения. В конце концов, принятие решения о том, когда вкладывать усилия, основывается на руководящих принципах, изложенных ниже.

1. Большое конкурентное преимущество/большие затраты усилий: эти контексты разрабатываются и внедряются собственными силами и на них тратится больше всего времени.
2. Небольшое преимущество/большие затраты усилий: покупаем!
3. Небольшое преимущество/небольшие затраты усилий: отличные задания для стажеров.
4. Другие комбинации как лотерея, и по ним решения должны приниматься коллегиально.



Последний этап, конкурентный анализ, не является частью первоначального процесса Event Storming Брандолини и был предложен Грегом Янгом для определения приоритетов предметных областей в DDD в целом. Мы считаем его полезной и забавной практикой, если к его выполнению подходить с юмором.

В целом процесс очень интерактивный, требущий участия всех присутствующих и обычно заканчивается весело. Чтобы все шло гладко, нужен опытный ведущий, но чтобы стать хорошим ведущим, не требуется прилагать столько же усилий, сколько необходимо, чтобы стать специалистом по ракетостроению (или экспертом по DDD). Прочитав эту книгу и проведя несколько пробных занятий для практики, вы сможете стать превосходным ведущим Event Storming!

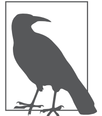
Желательно, чтобы ведущий следил за временем и имел план сеанса. Для четырехчасового сеанса приблизительное распределение времени будет выглядеть следующим образом:

- этап 1 (~30 минут): определение событий предметной области;
- этап 2 (~45 минут): создание временно́й линии;
- этап 3 (~60 минут): обратное повествование и идентификация команд;
- этап 4 (~30 минут): определение агрегатов/ограниченных контекстов;
- этап 5 (~15 минут): конкурентный анализ.

Если вы заметили, что этапы выше в сумме дают меньше 4 часов, то имейте в виду, что в процессе вы должны будете сделать несколько перерывов для участников, а также оставить себе время для подготовки пространства и ознакомления участников с правилами.

Представляем универсальную формулу определения размера

Ограниченные контексты — потрясающая отправная точка для определения оптимального размера микросервисов. Но имейте в виду, что границы микросервисов не всегда совпадают с границами ограниченных контекстов в DDD или Event Storming. На самом деле границы микросервисов могут меняться с течением времени и, как правило, следуют растущей детализации микросервисов с развитием организаций и приложений, частью которых они являются. Например, Адриан Кокрофт (https://oreil.ly/AzK4h) отметил, что это определенно повторяющаяся тенденция, которую они наблюдали в Netflix (https://oreil.ly/LXK8F).



Никто не определяет идеальные границы микросервисов с первой попытки

Даже в успешных проектах внедрения микросервисов команды начинают не с сотен, а с гораздо меньшего количества микросервисов, близко соответствующих ограниченным контекстам. С течением времени команды делят микросервисы, сталкиваясь с зависимостями координации, которые им необходимо устранить. Это также означает, что от команд не ожидается, что они будут «правильно» определять границы сервисов. Вместо этого границы изменяются с течением времени и постепенным движением в сторону большей детализации.

Стоит отметить, что обычно проще разделить сервис, чем снова объединить несколько сервисов или перенести возможности из одного сервиса в другой. Это еще одна причина, почему мы рекомендуем начинать с обобщенного, крупномодульного проекта и ждать, пока не появятся дополнительные знания о предметной области и не будет достигнут достаточно высокий уровень сложности, прежде чем разделить сервис и увеличить детализацию.

Мы выявили три принципа, касающиеся детализации микросервисов, и назвали их универсальной формулой определения размера микросервисов.

Универсальная формула определения размера. Чтобы прийти к микросервисам разумного размера:

- начинайте с нескольких микросервисов, возможно, совпадающих с ограниченными контекстами;
- продолжайте процесс деления с ростом вашего приложения и сервисов, руководствуясь потребностями предотвращения координации;
- придерживайтесь курса на снижение координации. Это гораздо важнее, чем текущее понимание «идеального» размера сервиса.

Резюме

В этой главе мы обсудили важный вопрос правильного определения размера микросервисов, рассмотрели предметно-ориентированное проектирование, популярную методологию декомпозиции сложных систем, объяснили высокоэффективный процесс анализа предметной области с помощью методологии Event Storming и представили универсальную формулу определения размера микросервисов, которая предлагает уникальные рекомендации, как делать это эффективно.

В следующих главах мы углубимся в реализацию и покажем, как управлять данными в слабосвязанной многокомпонентной среде микросервисов. Вдобавок мы познакомим вас с примером реализации нашего демонстрационного проекта: системой онлайн-бронирования.

ГЛАВА 5

Работа с данными

В этой главе мы расскажем, почему микросервисы должны «владеть своими данными» и что это означает для вашей архитектуры. Обсудим, когда и как использовать наиболее важные шаблоны управления данными в микросервисах: делегаты, хранилища озера данных, саги (Saga), Event Sourcing (регистрация событий) и разделение ответственности на команды и запросы (command query responsibility segregation, CQRS). Обсуждая эти важные темы, мы попытаемся продемонстрировать их на практических примерах, используя наш проект.

Одна из первых проблем в практической разработке микросервисов, с которой сталкиваются почти все, — работа с данными. Если бы не многочисленные проблемы управления данными в этом пространстве, то превратить сложные монолитные реализации в слабосвязанные управляемые микросервисы «небольшого размера» было бы довольно легко.

Подходы к проектированию логических и физических моделей при реализации микросервисов отличаются от подходов, применяемых при проектировании таблиц данных для обычных *N*-уровневых монолитных приложений. В этой главе мы увидим, почему возникают различия, какие шаблоны используются при разработке микросервисов и какие методы помогают решать дополнительные сложности, возникающие при реализации систем микросервисов.

Возможность независимого развертывания и обмена данными

В главе 4 мы отметили, что, по мнению Сэма Ньюмана (http://shop.oreilly.com/product/0636920033158.do), микросервисы должны быть:

- слабо связанными друг с другом, что также означает возможность независимого развертывания;

- плотно связными внутри, то есть содержать элементы, интенсивно взаимодействующие между собой.

Когда сервисы слабо связаны, изменение одного сервиса не должно приводить к изменению другого. Возможно, вы помните, что основное преимущество архитектуры микросервисов — увеличение скорости в сочетании с безопасностью и возможностью масштабирования. Это преимущество достигается за счет устранения или по крайней мере уменьшения потребности в координации между микросервисами. Одним из важных аспектов слабой связанности является то, что мы называем *возможностью независимого развертывания*, — это возможность вносить изменения в один микросервис и развертывать его без необходимости изменять или развертывать какие-либо другие части системы, любые другие микросервисы. Это действительно важно и становится очевидным, если представить, как выглядит типичный конвейер развертывания в архитектуре микросервисов. На рис. 5.1 вы можете увидеть упрощенное графическое представление конвейеров развертывания для нескольких микросервисов, проходящих через несколько сред на пути к промышленной среде.

Рис. 5.1. Пример конвейера выпуска для микросервисов, работающих в нескольких средах

Процесс передачи кода через конвейер развертывания становится значительно более сложным и хрупким, если развертывание одного микросервиса вызывает волновые эффекты, связанные с необходимостью повторного развертывания других частей приложения. Такая взаимозависимость может поставить под угрозу как скорость, так и безопасность всей системы. С другой стороны, имея возможность развертывать микросервисы независимо, не беспокоясь о волновых эффектах, мы сможем обеспечить оперативность и безопасность развертывания.

Есть множество причин, мешающих организовать независимое развертывание микросервисов, но в контексте управления данными наиболее типичной является совместное владение пространством данных несколькими микросервисами. Оно может помешать достичь слабой связанности и развертывать код независимо.

В следующих разделах мы начнем изучение методов предотвращения совместного использования данных с обсуждения понятия владения данными микросервисами.

Микросервисы владеют своими данными

В монолитных архитектурах обмен данными — обычная практика. В типичных устаревших системах, а также в системах с более модульной сервис-ориентированной архитектурой (service-oriented architecture, SOA) несколько сервисов совместно владеют одними и теми же данными, и это считается нормальным. На самом деле такое положение вещей вполне ожидаемо: общие данные — основной способ интеграции различных модулей в крупной системе. Иногда, когда мы говорим о «монолите», люди воображают, что у него нет модульности и это нечто большое, не имеющее деления на какие-либо компоненты. Это не соответствует истине. Разработчики давно знают, что разделение большой кодовой базы на более мелкие части очень полезно для организации кода и управляемости. Но ключевым недостатком до появления микросервисов было отсутствие возможности независимого развертывания модулей, составляющих монолиты. Это делало их сильно связанными в отношении затрат на координацию! Показательный пример: в первую очередь из-за связанности по данным проекты SOA никогда не имели возможности независимого развертывания сервисов и, следовательно, способности работать быстро и безопасно в больших масштабах.

Рассмотрим пример, как может возникнуть проблема. Предположим, что несколько микросервисов совместно владеют таблицей клиентов в базе данных, как показано на рис. 5.2, в подразделе «Владение данными и шаблон делегирования данных» далее в главе. Под владением мы подразумеваем чтение и изменение данных в общей таблице несколькими различными микросервисами.

Представьте, что микросервису поиска авиабилетов необходимо изменить тип одного из столбцов в общей таблице. Если разработчики этого микросервиса просто реализуют это, скажем, поменяв тип `integer` на `float` или что-то в этом роде, то изменение может привести к нарушению работы микросервисов бронирования или отслеживания рейсов, поскольку они тоже обращаются к той же таблице и могут полагаться на получение из этого поля значений определенного

типа. Чтобы избежать ошибок из-за изменения модели данных в угоду микросервису поиска авиабилетов, нам также необходимо соответствующим образом изменить код микросервиса бронирования и, возможно, других, а затем придется повторно развернуть все измененные микросервисы вместе. Волновые эффекты, вызванные изменением уровня данных, очень распространены, когда несколько компонентов совместно владеют данными, и могут вызвать значительную связанность различных сервисов, что может стать проблемой для возможности независимого развертывания.

Совместное использование данных — основной фактор, препятствующий независимой разработке и независимому развертыванию в монолитах. Напротив, в архитектуре микросервисов возможность независимого развертывания считается основной ценностью. Следовательно, совместное использование данных запрещено — микросервисам не разрешается совместно владеть данными. В базе данных должны быть проведены четкие границы, отделяющие наборы данных, принадлежащие разным микросервисам, или, как мы обычно формулируем принцип: микросервисы должны владеть своими (встроенными) данными.

Хотя владение данными — универсальное правило для микросервисов, этот принцип подразумевает несколько важных нюансов, которые важно четко понимать. В следующем подразделе мы обсудим одно из таких соображений более подробно.

Владение данными не должно приводить к резкому увеличению количества кластеров базы данных

При создании сложных приложений часто приходится иметь дело с различными типами баз данных. Наборы данных в них (например, «таблицы» в реляционных БД) не должны иметь нескольких совладельцев. Большие системы могут состоять из сотен микросервисов. Означает ли это, что мы должны развернуть сотни отдельных кластеров Cassandra, Postgres, Redis или MySQL? Командам, внедряющим микросервисы, нужна ясность в отношении того, как далеко они должны заходить в интерпретации понятия «микросервисы должны владеть своими данными». Базы данных сами по себе — сложные программные системы. Они не развертываются только на одном сервере. Часто они развертываются не на одном, а на множестве серверов, чтобы обеспечить резервирование, надежность и масштабируемость, — возможно, на десятках серверов в разных географических регионах. Когда мы говорим об идее владения данными, команды задаются вопросом, нужно ли им создавать массивные кластеры баз данных для каждого микросервиса.

Это может превратиться в серьезную проблему. Если бы архитектура микросервисов требовала огромного количества кластеров баз данных (по одному или

более на микросервис), то это был бы самый затратный архитектурный стиль в истории нашей отрасли (или близкий к нему). Независимость по данным не означает, что для каждого микросервиса должен развертываться свой кластер масштабируемых, избыточных и сложных баз данных.

КЛЮЧЕВОЕ РЕШЕНИЕ: МИКРОСЕРВИСЫ МОГУТ СОВМЕСТНО ИСПОЛЬЗОВАТЬ ФИЗИЧЕСКИЕ КЛАСТЕРЫ БАЗ ДАННЫХ

Микросервисы могут совместно использовать физические кластеры баз данных. Пока сервисы не используют одно и то же логическое табличное пространство и не изменяют одни и те же данные, совместное использование физических баз данных является нормальным.

Независимость по данным заключается скорее в отсутствии пересечений потоков данных, чем в чем-то другом. Речь идет о возможности развертывания микросервисов с разными базами данных, если это необходимо. Но совершенно необязательно развертывать каждый сервис с отдельным кластером БД. Стоимость — важный фактор, как и простота. До тех пор, пока несколько микросервисов не используют (и, что особенно важно: не изменяют) одни и те же данные, требование независимости по данным считается удовлетворенным.

Владение данными и шаблон делегирования данных

Рассмотрим пример в контексте нашей системы онлайн-бронирования. Сначала рассмотрим пример системы с обычной монолитной *N*-уровневой архитектурой. Такое приложение все равно будет разделено на разные, более мелкие модули. Они могут даже развертываться как сетевые сервисы. И определенно могут быть достаточно маленькими, чтобы их можно было назвать «микро». Однако это не обязательно означает, что они являются микросервисами. Эти модули могут считаться микросервисами только в том случае, если их модульность обусловлена устранением координации и, более конкретно, если они слабо связаны и могут развертываться независимо. Если сервисы разделены произвольно и тесно связаны, то мы не можем назвать такую систему примером архитектуры микросервисов.

В нашем сценарии, изображенном на рис. 5.2, есть три сервиса, использующие данные из таблицы «рейсы»: поиск рейсов, управление бронированием и отслеживание рейсов.

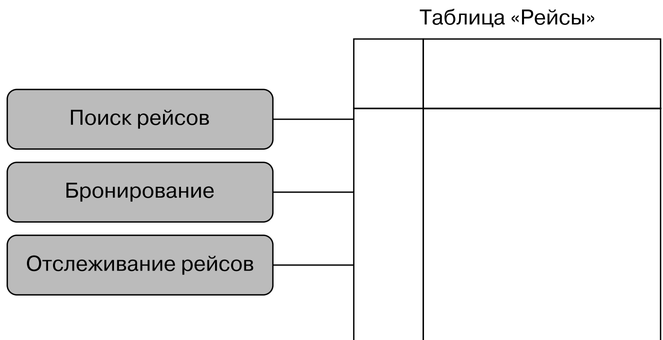


Рис. 5.2. Пример монолитного управления данными, характеризующегося их совместным использованием

Очевидно, что с точки зрения предыдущего анализа в этой главе, такая организация данных проблематична для архитектуры микросервисов, поскольку три сервиса совместно используют одни и те же данные и это мешает возможности независимого развертывания.

Как исправить эту ситуацию? Эту конкретную проблему на самом деле довольно легко решить, использовав простой прием скрытия общих данных за сервисом делегирования, как показано на рис. 5.3.

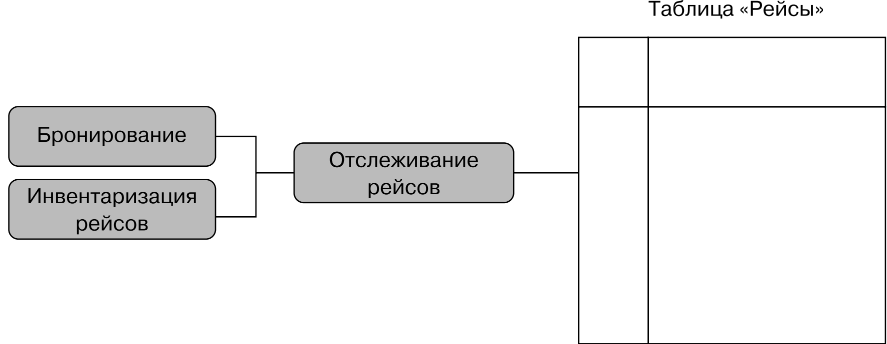


Рис. 5.3. Простое графическое представление скрытия данных с помощью делегата

По сути, здесь просто добавлен сервис «Перечень рейсов», поставляющий всю информацию о рейсах. Кроме того, любой сервис, которому требуется инфор-

мация о рейсах или нужно обновить информацию о рейсах, должен вызывать соответствующую конечную точку в сервисе перечня рейсов. Если реализовать достаточно гибкий вызов API поиска рейсов в сервисе перечня рейсов, то прежний сервис поиска рейсов просто станет частью нового сервиса. Что еще более важно, это позволит прекратить доступ к таблице рейсов непосредственно из сервисов бронирования и отслеживания рейсов. Любую необходимую информацию о рейсе они смогут получать из сервиса перечня рейсов.

Например, когда системе бронирования необходимо узнать, достаточно ли мест осталось на рейсе, она отправит соответствующий запрос в сервис перечня рейсов вместо того, чтобы запрашивать таблицу рейсов непосредственно в базе данных. Или, когда сервису отслеживания рейсов необходимо узнать или обновить местоположение самолета в полете, он снова сделает это через сервис перечня рейсов, а не путем прямого доступа и изменения таблицы рейсов. Таким образом, наш новый сервис может быть делегатом, который скрывает данные за собой, инкапсулирует их и оборачивает собой. Это прекратит совместное использование одной и той же таблицы данных несколькими сервисами.

Пожалуйста, обратите внимание, что в этом шаблоне, когда нескольким сервисам требуется доступ к одним и тем же данным, нам не обязательно преобразовывать один из этих сервисов в делегат. В предыдущем решении мы преобразовали сервис поиска авиабилетов в сервис инвентаризации и сделали так, чтобы он инкапсулировал таблицу рейсов. Вместо этого мы могли бы ввести новый сервис, например «Перечень рейсов» (Flight inventory), и заставить микросервис поиска авиабилетов ссылаться на него точно так же, как это делают сервисы бронирования и отслеживания.

Подход с добавлением делегата очень элегантен и с успехом может применяться во многих случаях. К сожалению, не все потребности в обмене данными можно удовлетворить таким образом. Было бы крайне наивно думать, что шаблон, который мы только что обсудили, подходит для всех сценариев. Существуют случаи, когда определенная функциональность обоснованно требует доступа или изменения данных напрямую. Примеры таких потребностей можно найти среди прочего в области аналитики, аудита данных и машинного обучения. Традиционные подходы к операциям с базами данных также требуют блокировки общих данных.

К счастью, существуют разумные решения и для таких вариантов, которые также позволяют избежать совместного использования данных. Чтобы понять решения в этой области, сначала рассмотрим разные типичные модели доступа к данным.

Дублирование данных для решения проблемы независимости

Когда распределенные данные только читаются, но не изменяются, как в контексте корпоративной аналитики, машинного обучения, аудита и т. д., общим решением является копирование наборов данных соответствующих микросервисов в общую область, которую обычно называют *озером данных* (data lake). Обратите внимание, что данные копируются, а не перемещаются! Озера данных — это накопители данных, доступные только для запросов на чтение. Микросервисы по-прежнему остаются авторитетными источниками соответствующих наборов данных и выступают в роли основных владельцев. Они просто передают соответствующие данные в озеро данных, где те накапливаются и становятся доступными для запросов. Ради целостности данных и ясности их происхождения важно *никогда* не обновлять подобные данные в агрегатном индексе, таком как озеро данных. Озера данных никогда не должны рассматриваться как базы данных записей. Они являются справочными хранилищами данных. В общем виде эта конфигурация изображена на рис. 5.4.

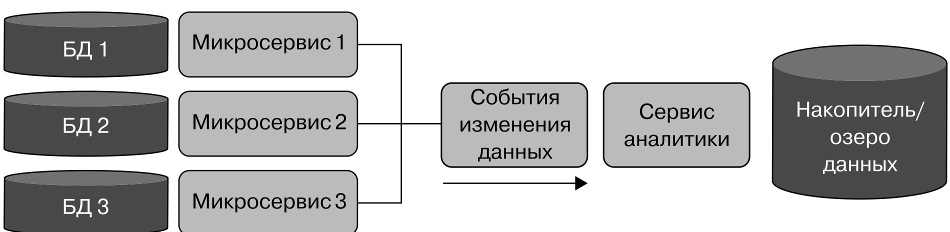


Рис. 5.4. Потоковая передача данных из микросервисов в озера данных

После передачи данных из хранилищ систем записей (system of record, SOR) микросервисов в накопители совокупные данные индексируются для оптимальной обработки запросов. Потоковая передача данных из SOR в озера данных обычно осуществляется с помощью надежной инфраструктуры обмена сообщениями. IBM MQ и RabbitMQ уже много лет используются в этом контексте. Kafka (https://kafka.apache.org) кажется самым популярным в настоящее время решением, в то время как Apache Pulsar (https://pulsar.apache.org), вероятно, самый выдающийся и интересный новичок в этой области.

Озера данных и общие индексы данных позволяют решить многие проблемы в сценариях, когда данные используются только для чтения. Но что делать, когда распределенные данные должны быть доступны не только для чтения? В следующем подразделе мы рассмотрим решение для случаев, когда необходимо

слаженным образом изменять данные в наборах, принадлежащих нескольким микросервисам, и обсудим реализацию распределенных транзакций в экосистеме микросервисов.

Распределенные транзакции и защита от сбоев

Рассмотрим пример из нашего проекта онлайн-бронирования. В частности, проанализируем, что происходит, когда кто-то бронирует место на рейсе, и для этого нам нужно выполнить *распределенную транзакцию* — скоординированное обновление данных, принадлежащих нескольким микросервисам, которые отвечают, например, за использование миль лояльности для оплаты, резервирование посадочного места и отправку маршрута на электронную почту клиента. Такие транзакции охватывают несколько микросервисов: в данном случае реализующих обработку платежей (с баллами лояльности), бронирование и рассылку уведомлений. Самое важное для нас — либо должны выполниться все три шага, либо не должен выполниться ни один из них. Например, может внезапно обнаружиться, что запрошенное место уже занято, — в начале процесса списания миль в счет оплаты место было доступно, но к моменту окончания процесса кто-то уже забронировал его. Очевидно, что мы не можем зарезервировать его дважды, поэтому стоит подумать, что делать в такой ситуации. В достаточно нагруженной системе такие состояния гонки и сбои неизбежны, поэтому, когда они происходят, нужно откатить весь процесс. Нам точно нужно как минимум вернуть баллы лояльности. Давайте разберемся, как координировать такую распределенную транзакцию.

В обычных монолитных приложениях подобным процессом можно безопасно управлять с помощью транзакций базы данных, которые, как утверждается, демонстрируют характеристики безопасности ACID даже в случае сбоев. Аббревиатура *ACID* расшифровывается как Atomicity, Consistency, Isolation, Durability — атомарность, согласованность, изоляция и надежность. Эти характеристики определяются следующим образом.

- *Атомарность*. Этапы в транзакции выполняются по принципу «все или ничего»: либо выполняются все, либо не выполняется ни один.
- *Согласованность*. Любая транзакция должна переводить систему из одного допустимого состояния в другое.
- *Изоляция*. Параллельное выполнение различных транзакций должно приводить к тому же конечному состоянию, как если бы транзакции выполнялись последовательно.

- *Устойчивость*. Как только транзакция будет зафиксирована (полностью выполнена), данные не могут быть утеряны, несмотря на любые возможные сбои.

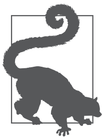
Микросервисы упрощают безопасное масштабирование систем. Однако важно уточнить: это не означает, что вы можете каким-то образом предотвратить возникновение сбоя. Полностью избежать сбоев с помощью микросервисов или любыми другими путями — невыполнимая задача. Сбои всегда будут происходить в мало-мальски сложной системе. Но мы можем учесть вероятность их появления и предусмотреть механизм для автоматического восстановления. В обычном управлении данными транзакции ACID (https://oreil.ly/B1OTU) — отличный пример такого мышления. Системы, реализующие транзакции ACID, предполагают, что сбои всех видов происходят постоянно, поэтому мы проектируем наши системы хранения данных так, чтобы сделать их устойчивыми к сбоям.

К сожалению, транзакции ACID плохо подходят для распределенных систем, где функциональность распределена между несколькими микросервисами, развернутыми в сети независимо друг от друга. Транзакции ACID обычно основаны на использовании эксклюзивных блокировок. Учитывая, что микросервисы монопольно владеют своими данными и не могут манипулировать данными, принадлежащими другим микросервисам, реализация таких блокировок была бы либо невозможной, либо очень затратной для системы микросервисов. Вместо этого нужно использовать шаблоны, созданные для распределенных архитектур. Далее мы представим популярное решение такого типа — распределенные транзакции и шаблон саг.

Распределенные транзакции и шаблон саг

Саги (Sagas) впервые были описаны Гектором Гарсия-Молиной (https://oreil.ly/bIRr3) в 1987-м, задолго до появления современных распределенных систем, и позже популяризированы Клеменсом Вестерсом в его блоге в 2012-м (https://oreil.ly/f5cLI) как эффективное решение для распределенных систем.

В сагах каждый шаг транзакции не только выполняет требуемое действие, но и определяет компенсирующее действие, которое должно выполняться, если потребуется откатить транзакцию из-за сбоя. Указатель (например, информация об обнаружении в очереди) на это компенсирующее действие регистрируется в *маршрутном листе (routing slip)* и передается на следующий шаг. Если один из последующих шагов завершается неудачей, то он запускает выполнение всех компенсирующих действий в маршрутном листе, тем самым «отменяя» изменения и приводя систему в разумно компенсированное состояние.



Шаблоны саг не являются прямыми эквивалентами транзакций ACID

Саги не гарантируют возврат системы в исходное состояние при откате распределенной транзакции, но стараются привести ее в рациональное состояние, отражающее приемлемый уровень отмены частично выполненной транзакции.

Чтобы понять, что мы подразумеваем под «рациональным состоянием», посмотрим на первоначальный пример бронирования места, изображенный на рис. 5.5.

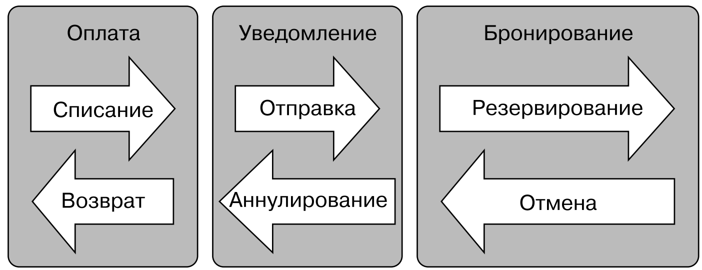
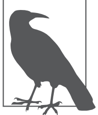


Рис. 5.5. Транзакция, распределенная между несколькими микросервисами

Если по какой-либо причине попытка бронирования не удалась, то оно будет отменено, но при этом будут применены компенсирующие действия для предыдущих шагов: уведомления и оплаты. Компенсирующее действие этапа оплаты возвращает деньги клиенту. В зависимости от типа платежа возврат может быть обработан не сразу. Поэтому система может не сразу вернуться в исходное состояние, но в конечном итоге клиент получит все свои деньги. Клиент заметит два платежных действия, отменяющих (компенсирующих) друг друга, в отличие от транзакций ACID, целью которых является незаметность следов отмены транзакции.

В случае отправки уведомлений ситуация может стать еще более запутанной. Может не получиться буквально отозвать отправленное электронное письмо или текстовое сообщение, поэтому компенсирующая транзакция может заключаться в отправке нового сообщения, уведомляющего клиента о том, что предыдущее сообщение следует игнорировать и бронирование фактически было неудачным. В некоторых обстоятельствах это решение может быть разумным (для других случаев, когда такое решение не годится, мы покажем правильную последовательность далее в главе), но не возвращает систему в исходное состояние: клиент получит два сообщения вместо того, чтобы не получить ни одного.

Мы надеемся, что эти два примера дали четкое представление о некоторых отличиях компенсирующих транзакций в сагах от обычных транзакций ACID.



Последовательность событий в сагах имеет значение

Обратите внимание, что последовательность событий в сагах действительно имеет значение и требует особого внимания при конструировании. Шаги, которые труднее компенсировать, обычно лучше размещать ближе к концу транзакции. Например, если это позволяют бизнес-правила, отправка уведомления в самом конце процесса может избавить от необходимости отправлять много сообщений с исправлениями. Таким образом, к тому времени, когда транзакция дойдет до отправки уведомления, мы будем знать, что предыдущие шаги выполнены успешно.

Сервисы-делегаты, озера данных и саги — эффективные шаблоны. Они могут решить многие проблемы изоляции данных в архитектуре микросервисов, но не все. В следующем разделе мы обсудим мощный дуэт паттернов проектирования: Event Sourcing (регистрация событий) и CQRS (разделение ответственности на команды и запросы). Они способны решить практически все остальные проблемы, предоставляя полный набор инструментов для управления данными в среде микросервисов.

Event Sourcing и CQRS

До этого момента обсуждались способы избежать обмена данными при использовании традиционного реляционного моделирования данных. Мы показали, как можно решить некоторые проблемы с совместным использованием данных, но в особо сложных сценариях неизбежны случаи, когда реляционное моделирование не обеспечивает желаемого уровня изоляции данных и слабой связанности. Очень распространенный пример — когда командам необходимо создать «соединение» наборов данных, принадлежащих разным микросервисам. По своей сути реляционное моделирование основывается на таких основополагающих принципах, как нормализация данных, их повторное использование и перекрестные ссылки на общие элементы данных. То есть оно в корне смещено в пользу совместного использования данных.

Event Sourcing

Иногда вместо того, чтобы пытаться обойти предрасположенность к реляционному моделированию, лучше переключиться на совершенно другой способ моделирования данных. Подход к моделированию, позволяющий избежать

совместного использования данных и потому популярный в микросервисах, известен как Event Sourcing (регистрация события).

Одно из самых ранних упоминаний об Event Sourcing встречается в статье Мартина Фаулера 2005 года (https://oreil.ly/BHKl9). В 2014 году Грег Янг выступил с важным докладом[1] на конференции об Event Sourcing, вызвавшим всплеск популярности этого паттерна проектирования. Грег представлял важное мнение и был одним из ключевых пропагандистов в этой области. Мы должны быть ему очень благодарны за продвижение шаблона Event Sourcing (и его связь с CQRS, еще одной важной моделью, которую мы обсудим позже в этой главе). По словам Грега, Event Sourcing — это подход к моделированию данных, суть которого заключается в хранении событий, а не состояний предметных объектов системы:

> «Идея Event Sourcing заключается в сохранении фактов, а все “состояния” (структурные модели) преходящи и являются лишь первыми производными от этих фактов».

В этом контексте под фактами Янг подразумевает репрезентативные значения событий. Примером может служить такое событие: «цена места экономкласса на рейсе LAX-IAD увеличилась на 200 долларов».

Event Sourcing в бухгалтерии и шахматах

Не имеющим опыта применения Event Sourcing этот шаблон может показаться странным. Большинство людей, которые не занимались системами, работающими с высокочастотными торговыми платформами, или не имели большого опыта разработки микросервисов, вероятно, не имеют никакого опыта применения Event Sourcing. Тем не менее мы можем легко найти примеры систем Event Sourcing в реальной жизни. Если вы когда-нибудь видели бухгалтерский журнал, то знаете, что это классическое хранилище событий. Бухгалтеры регистрируют отдельные операции, а баланс является результатом суммирования всех операций. Бухгалтеры не записывают «состояние». То есть они просто записывают итоговый баланс после каждой операции, не фиксируя сами операции. Точно так же при игре в шахматы записывается шахматная партия: никто не записывает положение каждой фигуры на доске после каждого хода. Вместо этого записываются ходы по отдельности, и положение фигур на доске после каждого хода является результатом суммы всех сделанных ходов.

Например, рассмотрим запись первых семи ходов исторической шестой партии между многократным чемпионом мира по шахматам Гарри Каспаровым

[1] *Young G.* CQRS and Event Sourcing. Code on the Beach, 2014. https://oreil.ly/5-d5u.

и суперкомпьютером IBM Deep Blue в 1997 году. Представленная в алгебраической нотации (https://oreil.ly/txpy7) партия выглядит следующим образом.

1. e4 c6
2. d4 d5
3. Nc3 dxe4
4. Nxe4 Nd7
5. Ng5 Ngf6
6. Bd3 e6
7. N1f3 h6

Соответствующее состояние после первых семи ходов показано на рис. 5.6.

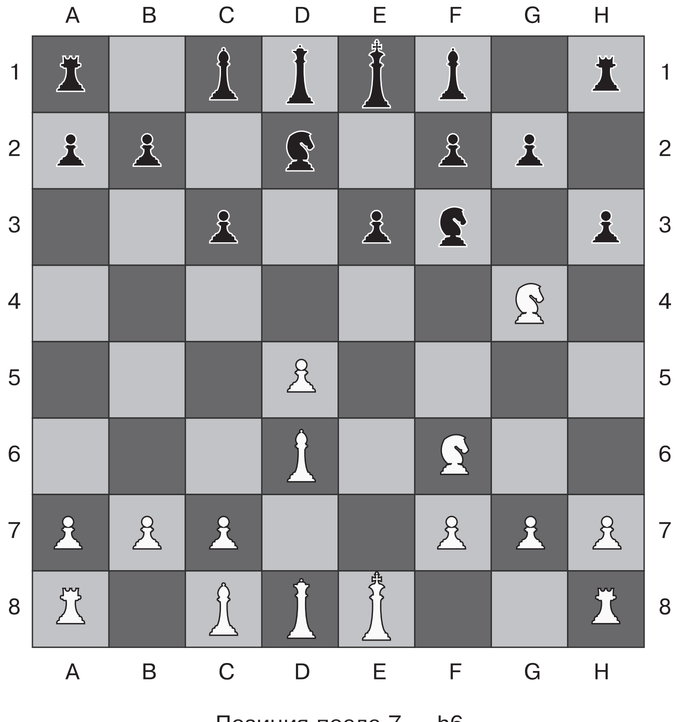


Рис. 5.6. Deep Blue против Каспарова (источник: Wikipedia (https://oreil.ly/-chbm))

Мы можем полностью воссоздать шахматную партию, например, между Каспаровым и Deep Blue, имея список всех ходов. Это аналоговый эквивалент Event Sourcing из реальной жизни.

Event Sourcing и реляционное моделирование

В обычных системах обработки данных, таких как реляционные базы данных или даже более современные документные базы данных и базы данных NoSQL, мы обычно храним состояние чего-либо. Например, текущую цену экономместа на рейсе. В Event Sourcing используется совершенно другой подход: хранится не текущее состояние, а *факты*, описывающие этапы изменения данных. Текущее состояние системы является производной, значением, которое вычисляется на основе последовательности изменений (событий).

Рассмотрим пример. Реляционная модель данных, описывающая систему управления клиентами для системы бронирования авиабилетов, может выглядеть как схема на рис. 5.7.

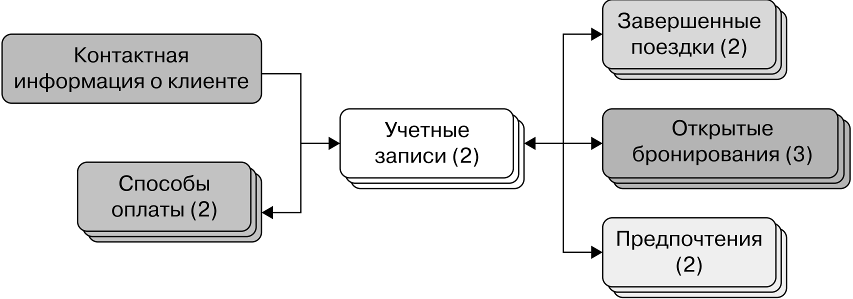


Рис. 5.7. Пример реляционной модели данных

Как видите, модель данных может состоять из таблицы с контактной информацией клиентов, связанной отношением «один ко многим» с таблицей с учетными записями клиентов и с таблицей со способами оплаты. В свою очередь, каждая запись в таблице с учетными записями клиентов (например, деловые или личные учетные записи) может ссылаться на несколько завершенных поездок, открытых бронирований и предпочтений. Детали могут различаться, но именно такую модель данных большинство инженеров-программистов разработало бы при использовании обычных баз данных.

Следуя шаблону Event Sourcing, ту же модель данных можно спроектировать в виде последовательности событий, как показано на рис. 5.8. Здесь показаны события, приводящие систему в то же состояние, которое было описано выше в модели, ориентированной на состояние: сначала была получена контактная информация клиента, затем открыт личный кабинет, после чего последовал выбор способа оплаты. После нескольких бронирований и поездок этот

клиент, по-видимому, решил также создать деловую учетную запись. Он добавил платежную информацию и начал бронировать поездки с помощью этой новой учетной записи. Попутно было также добавлено и изменено несколько предпочтений, что привело систему в то же состояние, которое было показано на рис. 5.7.

Но есть важное исключение: здесь можно видеть точную последовательность «фактов», которые привели к текущему состоянию, в отличие от простого просмотра результата в представлении, ориентированном на состояние.

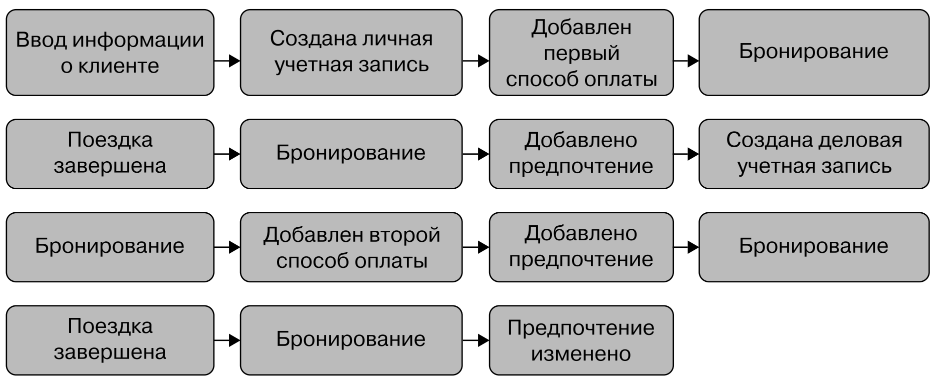


Рис. 5.8. Пример модели данных, основанной на источнике событий

Последовательность событий на диаграмме дает то же состояние, которое мы имели в реляционной модели данных. Это представление эквивалентно предыдущему, за исключением того, что выглядит очень, очень непохоже. Например, обратите внимание, что такое представление выглядит гораздо более однородным. Оно требует принять значительно меньше трудных решений, касающихся различных типов сущностей и их взаимоотношений. Модель Event Sourcing в некотором смысле намного проще, потому что вы имеете просто множество бизнес-событий и можете рассчитать текущее состояние как производную от них.

Модель Event Sourcing не только более простая и предсказуемая, в ней нет ссылочных отношений между различными сущностями. Если бы нам понадобилось разделить данные, то для этого достаточно было бы сказать, что каждый тип события принадлежит своему микросервису, и тем самым исключить совместное использование данных. Например, у нас может быть микросервис, представляющий демографическую информацию о клиентах, и тогда «ввод информации о клиенте» будет событием, которое относится к этой системе записей естественным образом.

Как выглядит событие

Теперь, получив, как мы надеемся, хорошее представление о шаблоне Event Sourcing и особенностях его работы, немного углубимся в моделирование данных и управление ими в Event Sourcing. Суть этого подхода заключается в регистрации последовательности событий. Состояние — это просто то, что вычисляется на основе событий, то есть состояние — это функция событий. Хорошо, это звучит немного математически, но как вообще выглядит событие? Что ж, события очень просты. Структура данных события имеет простую «форму» и состоит из трех частей.

Во-первых, каждое событие должно иметь какой-то уникальный идентификатор. Для этой цели можно использовать, например, универсально уникальные идентификаторы (universally unique identifier, UUID), поскольку они глобально уникальны, и эта уникальность весьма кстати в распределенных системах. Событие также должно иметь тип, чтобы мы не путали разные виды событий. И наконец, данные, относящиеся к этому типу события:

```
{
  "eventId" : "afb2d89d-2789-451f-857d-80442c8cd9a1",
  "eventType" : "priceIncreased",
  "data" : {
    "amount" : 120.99,
    "currency" : "USD"
  }
}
```

При работе с событиями технические проектные решения намного проще. Большая часть времени уходит на правильное описание полей событий, относящихся к предметной области, на основе бизнес-логики. Здесь гораздо меньше субъективизма, чем при формировании таблиц и отношений в реляционном подходе.

Вычисление текущего состояния с помощью проекций

Как на самом деле вычисляется состояние чего-либо в определенный момент времени (например, текущее)? Для этого мы запускаем то, что в Event Sourcing называется *проекциями*. Проекции дают состояние, основанное на событиях, и тоже имеют довольно простую реализацию. Чтобы запустить проекцию, нужна функция проекции. Она принимает текущее состояние и новое событие для вычисления нового состояния.

Например, функция проекции `priceUp`, вычисляющая увеличенную цену авиабилета, может выглядеть следующим образом:

```
function priceUp(state, event) {
  state.increasePrice(event.amount)
}
```

Она в какой-то мере эквивалентна SQL-запросу `UPDATE prices SET price=…` в реляционной модели. Если бы у нас также была соответствующая функция проекции, вычисляющая сниженную цену, и мы хотели бы рассчитать цену (состояние) в какой-то момент, то могли бы запустить проекцию, вызвав функции проекции для всех соответствующих событий, например такие:

```
function priceUp(state, event) {
  state.increasePrice(event.amount)
}

function priceDown(state, event) {
  state.decreasePrice(event.amount)
}

let price = priceUp(priceUp(priceDown(s,e),e),e);
```

Имеющие опыт функционального программирования могут заметить, что текущее состояние — это *левая свертка* событий, происшедших до текущего времени. Обратите внимание, что с помощью Event Sourcing можно вычислить не только текущее состояние, но и состояние на любой момент времени. Эта функциональность открывает бесконечные возможности для сложной аналитики, позволяющей задавать вопросы типа: «Хорошо, я знаю текущее состояние объекта, но каково было это состояние в некоторый момент в прошлом?» Эта гибкость может стать одним из сильных преимуществ использования Event Sourcing, если вам часто приходится отвечать на подобные вопросы.

Повышение производительности с помощью скользящих снимков

Здесь следует отметить одну вещь: проекции могут быть затратными с точки зрения вычислений. Если искомое значение текущего состояния является результатом последовательности тысяч изменений состояния, например баланса банковского счета, то хотели бы вы всякий раз выполнять вычисления с нуля, чтобы получить нужное значение текущего баланса? Вы могли бы отметить, что такой подход может привести к пустой трате времени и вычислительных ресурсов. Этот процесс не может быть таким же мгновенным, как простое получение текущего состояния. И вы были бы правы. Однако мы можем оптимизировать скорость, не изменяя сам подход. Вместо того чтобы пересчитывать все с самого начала — например, с момента открытия банковского счета, — можно продолжать сохранять промежуточные значения по ходу дела, а позже быстро рассчитать состояние, начав с предшествующего снимка. Это значительно ускорило бы вычисления.

В зависимости от реализации хранилища событий промежуточные снимки обычно создаются в различные моменты времени. Способ выбора подходящего момента времени для очередного снимка может зависеть от предметной области вашего приложения. Например, в банковской системе моментальные снимки остатков на счетах можно создавать в последний день каждого месяца, так что если вам понадобится баланс на 15 января 2020 года, то вам достаточно будет начать со снимка, сделанного 31 декабря 2019 года, и рассчитать проекцию за последующие две недели, а не за весь период существования банковского счета.

В Event Sourcing сохраненные проекции обычно называются *скользящими снимками*. Особенности реализации скользящих снимков и проекций могут зависеть от контекста приложения. Так, ранее использованные ежемесячные скользящие снимки в примере банковского приложения имели определенный смысл, поскольку такой подход тесно связан с происходящим в реальной жизни. Банки рассчитывают различные типы остатков на конец месяца, квартала и года. Такая операция известна как «закрытие бухгалтерских книг». Вы всегда должны стараться находить *естественные временные точки* в собственных предметных областях и согласовывать свои снимки с ними.

Позже в этой главе мы увидим, что паттерн, называемый разделением ответственности на команды и запросы (CQRS), позволяет гораздо больше, чем простое кэширование состояния в скользящих снимках.

Теперь, получив представление об Event Sourcing, давайте посмотрим, как его реализовать. Как будет выглядеть само хранилище событий? И как бы мы его организовали? Ответим на эти вопросы в следующем подразделе.

Хранилище событий

Хранилища событий могут быть относительно простыми системами. Для их реализации можно использовать различные системы хранения данных. Простые файлы в файловой системе, корзины Amazon Simple Storage Service (S3) или любые базы данных, способные надежно хранить последовательности записей, — все они могут использоваться для этой цели. Интерфейс хранилища событий должен поддерживать три основные функции:

- сохранять новые события в правильной последовательности, чтобы мы могли извлекать события в том порядке, в каком они были сохранены;
- уведомлять подписчиков, которые создают проекции, о новых интересующих их событиях, и поддерживать паттерн «Конкурирующие потребители» (https://oreil.ly/WZ9Ss);

- возвращать N событий определенного типа после события X для сверки, то есть для перерасчета на случай потери проекцией, возникновение угрозы или сомнений.

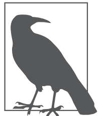
Итак, по сути, базовый интерфейс хранилища событий состоит всего из двух функций:

```
save(x)
getNAfterX()
```

Кроме того, должна иметься надежная система уведомлений, позволяющая потребителям подписываться на события. Под «надежностью» мы подразумеваем соответствие паттерну «Конкурирующие потребители». Он важен, поскольку в любой системе, создающей проекции событий, может потребоваться поддерживать несколько экземпляров клиентов, «прослушивающих» события, как для избыточности, так и для масштабируемости. Система уведомлений должна обеспечивать однократную доставку одному экземпляру клиента, чтобы избежать случайного дублирования событий, приводящих к повреждению данных. Здесь можно использовать два подхода.

1. Применить готовую реализацию очереди сообщений, предоставляющей такие гарантии, например Apache Kafka (https://kafka.apache.org).
2. Разрешить потребителям регистрировать конечные точки HTTP для обратных вызовов. В таком случае система уведомлений может просто вызывать конечные точки для каждого нового события и переложить ответственность за распределение работы на балансировщика нагрузки на стороне потребителя.

Ни один из подходов, по сути, не является лучшим. Один из них относится к типу push-based, а другой — к типу pull-based, и в зависимости от потребностей вы можете предпочесть один другому.



Ознакомьтесь с примером реализации

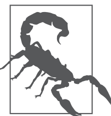
Во время написания этой книги мы опубликовали на GitHub (https://oreil.ly/LPD8y) эталонную реализацию скелета хранилища событий, чтобы вы могли ознакомиться с ней, попробовать запустить код или внести свой вклад.

Для реализации надежных проекций системы Event Sourcing часто используют дополнительный паттерн, известный как CQRS. В следующем подразделе мы рассмотрим идеи, лежащие в его основе, и попытаемся понять его суть.

Разделение ответственности на команды и запросы (CQRS)

Проекции для продвинутых систем Event Sourcing обычно строятся с использованием паттерна «Разделение ответственности на команды и запросы» (Command Query Responsibility Segregation, CQRS). Идея CQRS заключается в том, что способ, которым мы запрашиваем системы, и способ хранения данных не обязательно должны быть одинаковыми. Говоря о хранилище событий и о том, насколько простым оно может быть, мы упустили одну мысль: простой интерфейс функций `save(x)` и `getNAfterX()` не позволит нам выполнять сложные запросы. Например, этот интерфейс не позволит нам запросить все бронирования, в которых пассажиры обновили информацию о своем посадочном месте за последние 24 часа. Запросы такого рода должны обрабатываться за пределами хранилища событий, чтобы сохранить его простым и целенаправленным. Хранилище Event Sourcing должно решать только проблему авторитетного и надежного хранения журнала событий, а обрабатывать сложные запросы должна другая система, выступающая в роли подписчика хранилища событий, которая может создавать индексы, оптимизированные для извлечения данных любым необходимым способом. Следуя идее, лежащей в основе CQRS, вы не должны пытаться решать проблемы хранения данных, управления ими и обработки запросов в одной и той же системе. Эти проблемы должны решаться независимо друг от друга.

Большим преимуществом Event Sourcing и CQRS является возможность разрабатывать очень детализированные слабосвязанные компоненты. С помощью Event Sourcing можно создавать крошечные микросервисы, управляющие только одним типом событий или генерирующие один отчет. Целенаправленное использование Event Sourcing и CQRS может вывести нас на более высокий уровень детализации архитектуры микросервисов. Как таковые, они играют решающую роль в архитектурном стиле.



Не злоупотребляйте паттернами Event Sourcing и CQRS, они не панацея от всех проблем

Будьте осторожны, не злоупотребляйте паттернами Event Sourcing и CQRS. Используйте их только при необходимости, так как они могут усложнить вашу реализацию. Их не следует применять в качестве единственного подхода к моделированию данных для всей вашей системы. Все еще существует множество случаев, когда традиционная реляционная модель оказывается намного проще и предпочтительнее.

Event Sourcing и CQRS могут помочь избежать совместного использования данных микросервисами в сложных случаях, когда требуется организовать соединение данных через границы сервисов. Всегда рассматривайте другие, более простые подходы, такие как сервисы-делегаты, описанные выше в этой главе, прежде чем прибегать к Event Sourcing для реализации конкретного микросервиса.

Теперь, получив цельное, базовое представление об Event Sourcing и CQRS, давайте посмотрим, где еще можно и нужно использовать эти паттерны, кроме как для обеспечения слабой связанности микросервисов по данным.

Event Sourcing и CQRS за пределами микросервисов

Event Sourcing и CQRS, безусловно, могут иметь неоценимое значение для предотвращения совместного использования данных и обеспечения слабой связанности микросервисов. Однако их преимущества не ограничиваются слабой связанностью или даже архитектурой микросервисов. Event Sourcing и CQRS — это мощные инструменты моделирования данных, которые могут принести пользу самым разным системам.

Рассмотрим Event Sourcing и CQRS с позиций теоремы о согласованности, доступности и устойчивости к разделению (consistency, availability and partition tolerance, CAP). Эту теорему отлично сформулировал Эрик Брюер как гипотезу в своем докладе 2000 года (https://oreil.ly/hiQMB) на симпозиуме по принципам распределенных вычислений. Теорема в ее первоначальной форме утверждала, что любая распределенная система с общими данными может обладать только двумя из трех желаемых свойств, таких как:

- *согласованность* — наличие единого представления последнего состояния данных;
- *доступность* — возможность всегда получать или обновлять данные;
- *устойчивость к разделению* — получение точных данных даже при наличии сетевого разделения.

Со временем выяснилось, что не все комбинации CAP допустимы[1]. Работая с распределенной системой, необходимо учитывать устойчивость к разделению,

[1] *Brewer E.* CAP Twelve Years Later, InfoQ. 2012. https://oreil.ly/Pg1pO. Coda Hale. You Can’t Sacrifice Partition. 2010. https://oreil.ly/nHBoN.

поскольку сетевого разделения избежать невозможно и приходится жертвовать согласованностью или доступностью. Но что делать, если нам действительно нужно и то и другое? Было бы слишком по-детски настаивать на желании иметь все, если математически доказанная теорема (которой в итоге стала CAP) говорит, что это невозможно.

Но тут есть одна загвоздка! Теорема CAP говорит нам, что единая система с общими данными не может нарушать теорему. Но что, если с помощью CQRS мы создадим несколько систем и сведем к минимуму совместное использование данных? В таком случае мы сможем отдать приоритет согласованности в хранилище событий и доступности в индексах запросов. Конечно, это означает, что любая система, которую мы используем для индексации запросов, может привести к неправильной согласованности, но такие системы не являются авторитетными источниками и мы всегда можем переиндексировать данные из хранилища событий, если понадобится. В некотором смысле это действительно позволяет нам получить лучшее из обоих подходов.

Второе важное преимущество подходов Event Sourcing и CQRS связано с возможностью проверки. Используя реляционную модель данных, мы обновляем данные на месте. Например, если мы решим, что адрес или номер телефона клиента неверны, то обновим его в соответствующей таблице. Но что произойдет, если клиент позже оспорит свою запись? С реляционной моделью мы можем потерять историю изменений и оказаться беспомощными. В Event Sourcing у нас имеется полная история всех изменений, надежно хранящаяся, и можно увидеть, какими были значения в любой момент в прошлом, а также как и когда они изменялись.

Некоторые читатели могут заметить, что использование реляционной модели не обязательно означает потерю истории изменения данных. Они могут регистрировать каждое изменение в каком-либо файле или системах, таких как Splunk (https://oreil.ly/C3oY-) или ELK (https://oreil.ly/80teW). Итак, чем журналирование событий отличается от Event Sourcing? Не называем ли мы старое доброе журналирование новым модным именем? Ответ: однозначно нет. Все сводится к следующему: какая система является источником истины в нашей архитектуре? Кому можно «доверять», если содержимое журнала не соответствует текущему состоянию? В Event Sourcing «состояние» вычисляется по событиям, поэтому ответ очевиден. Для журналов Splunk это не так, поэтому вашим источником истины, скорее всего, будет реляционная модель, даже если время от времени сверять ее с журналами, чтобы выследить некоторые ошибки. Когда источником истины служит надежный журнал событий, вы используете подход Event Sourcing к моделированию данных. В противном случае он не будет источником истины независимо от того, сколько журналов вы создадите.

Резюме

В этой главе мы обсудили фундаментальную концепцию архитектуры микросервисов: изоляцию данных и принцип владения данными соответствующими микросервисами. Мы также исследовали, как этот принцип, даже обеспечивая слабую связанность и независимость развертывания, может привести к серьезным проблемам управления данными, если подходить к их решению с использованием традиционных методов моделирования данных, предназначенных для монолитных *N*-уровневых приложений. Кроме того, мы рассмотрели полный набор инструментов для решения описанных проблем в форме эффективных, проверенных и надежных паттернов, которые решают эти проблемы напрямую. И последнее, но не менее важное: мы представили новый подход к моделированию данных, заметно отличающийся от обычного реляционного моделирования. Объяснили преимущества и соответствующие контексты использования Event Sourcing и CQRS, в том числе за пределами микросервисов.

Вооруженные этими фундаментальными знаниями, мы теперь можем погрузиться в реализацию нашего примера проекта. Начнем с создания автоматизированной контейнерной инфраструктуры и конвейеров развертывания для проекта. Этот шаг имеет ключевое значение для решения операционных сложностей проекта, основанного на микросервисах. Затем мы поделимся подробными рекомендациями по созданию продуктивной и воспроизводимой рабочей среды разработчика — важнейшей основы для создания приятного опыта разработчика в гетерогенной среде. Наконец, мы попытаемся написать код для пары микросервисов в нашем проекте, используя все знания, полученные к этому моменту.

ГЛАВА 6

Создание конвейера инфраструктуры

В этой главе мы заложим основу нашей инфраструктуры и начнем с настройки учетной записи Amazon Web Services (AWS). После этого настроим инструмент под названием «конвейер непрерывной интеграции и непрерывного развертывания» (continuous integration and continuous delivery, CI/CD) для автоматизации развертывания изменений инфраструктуры. С помощью этих инструментов мы сможем определять и предоставлять инфраструктуры микросервисов на протяжении всей книги.

Ранее, в главе 2, мы создали команду разработки платформы, ответственную за создание инфраструктуры микросервисов. И решили, что она будет предлагать инфраструктуру как услугу. Это означает, что другие команды должны иметь возможность использовать инфраструктуру в режиме самообслуживания, не прибегая к интенсивной координации с командой разработки платформы. Включение модели «как услуга» требует определенных первоначальных затрат. Именно с этим помогут справиться инструменты, описанные в текущей главе.

Чтобы сократить объем работы, которую приходится выполнять командам микросервисов, нужно упростить перенос их кода с локальных рабочих станций в инфраструктуру, размещенную на хостинге. Соответственно, нужно облегчить командам создание сред и развертывание их сервисов в системе на хостинге. Мы должны сделать создание новой среды дешевым и простым, а также предоставить правильный набор инструментов, чтобы процесс выпуска новых версий был простым и безопасным.

На практике достичь этих целей затруднительно, если нет хорошего способа упростить внесение изменений в саму инфраструктуру. Сумев снизить затраты на создание и изменение инфраструктуры, мы сможем упростить создание

новых сред и уделить больше внимания совершенствованию инфраструктуры ради достижения целей, стоящих перед нашей системой.

К счастью, нам не нужно изобретать свое решение для передачи изменений в инфраструктуру. Мы можем позволить себе роскошь опереться на принципы и философию DevOps. В частности, использование практик DevOps для создания инфраструктуры как кода (infrastructure as code, IaC), CI и CD поможет нам добиться желаемого. Мы сможем быстрее, дешевле и безопаснее вносить изменения в инфраструктуру, а также масштабировать работу по созданию сред для наших команд разработки микросервисов.

DevOps и микросервисы

Цель DevOps — совершенствование способов разработки, выпуска и поддержки программного обеспечения. Для достижения этой цели может понадобиться уделить внимание организационному проектированию, культуре, процессам и подбору инструментария. Архитектура микросервисов преследует схожую общую цель, но добавляет дополнительные характеристики ограниченных сервисов и независимого развертывания и управления. Микросервисы и DevOps идут рука об руку — на самом деле было бы чрезвычайно сложно создавать приложения в стиле микросервисов, не применяя практики DevOps.

Внедряя практики DevOps, мы получаем возможность пользоваться преимуществами богатой экосистемы инструментов для управления кодом, сборкой и выпуском новых версий. Применение этих инструментов значительно сокращает время, необходимое для подготовки и запуска нашего инфраструктурного решения. К концу этой главы мы настроим облачные инструменты, которые сможем использовать для создания инфраструктуры микросервисов. Мы получим репозиторий IaC, начальный файл для кода инфраструктуры, конвейер тестирования и сборки (рис. 6.1) и облачную основу, в которой мы сможем создавать среды.

Рис. 6.1. Целевой конвейер

Но прежде, чем начать создавать конвейер, познакомимся с принципами и практиками DevOps, которые мы использовали для разработки проекта.

Принципы и практики DevOps

Создавая программное обеспечение с использованием методов DevOps, можно сократить время, необходимое для внесения изменений в приложения, без дополнительного риска. Соблюдая методологию, вы получаете возможность вносить изменения быстро и безопасно.

Это именно то преимущество, которое мы хотим предоставить с помощью цепочки инструментов инфраструктуры. Сумев повысить скорость и безопасность внесения изменений в инфраструктуру, мы сможем увеличить их количество, а значит сможем чаще вносить улучшения и предложить нашим командам разработки микросервисов более качественную услугу (в смысле платформу).

С этой целью реализуем в нашей инфраструктурной платформе три идеи из мира DevOps:

- неизменяемую инфраструктуру;
- инфраструктуру как код (infrastructure as code, IaC);
- непрерывную интеграцию и непрерывное развертывание (continuous integration and continuous delivery, CI/CD).

Рассмотрим каждую из этих идей более подробно, чтобы понять, как они нам помогут. Начнем с принципа неизменяемой инфраструктуры.

Неизменяемая инфраструктура

Объект является неизменяемым, если его нельзя изменить после создания. Единственный способ обновить неизменяемый объект — уничтожить его и создать новый. Неизменяемые объекты содержат поведение и структуры, которые легче предсказать и воспроизвести, поскольку они не меняются. Например, в программировании неизменяемый тип данных позволит присвоить значение при создании экземпляра, но не позволит изменить его. Если вы создали экземпляр данных `x` со значением `10`, то можете быть уверены, что он всегда будет иметь значение `10`. Такая предсказуемость может облегчить такие действия, как тестирование и репликация этих объектов.

Принцип неизменяемой инфраструктуры подразумевает применение свойства неизменяемости и к компонентам инфраструктуры. Предположим, мы должны настроить и установить сетевой балансировщик нагрузки с набором определенных маршрутов. Если применить принцип неизменности, то определенные нами сетевые маршруты нельзя изменить, не удалив балансировщик нагрузки и не создав новый.

Главное преимущество неизменяемости заключается в создании предсказуемой и легко воспроизводимой инфраструктуры. В традиционных системах операторам-людям приходится выполнять много работы вручную, чтобы все работало. Они устанавливают исправления, изменяют настройки, останавливают и запускают процессы. Серверы и устройства постоянно работают, и оператор формирует среду, в которой приложение сможет функционировать. Когда существует несколько сред и серверов, оператору приходится сформировать и настроить их все.

Но со временем, по мере внесения новых изменений (часто непоследовательных), состояние этих систем меняется. Становится все труднее поддерживать все серверы в одном и том же состоянии. Добавление новых серверов или внесение изменений в состояние среды становится проблемой из-за этой изменчивости и непредсказуемости. Непредсказуемость требует больше опыта и ручного труда, что замедляет развертывание и затрудняет предложение инфраструктурной платформы в виде инструмента самообслуживания, как мы описали в нашей операционной модели.

В таких случаях на помощь приходит неизменяемая инфраструктура. Приняв принцип неизменности, можно создать предсказуемую и легко воспроизводимую инфраструктуру. Это очень хорошо подходит для модели, на которую мы ориентируемся. Итак, примем это первое ключевое решение для нашей инфраструктуры.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИНЯТЬ ПРИНЦИП НЕИЗМЕНЯЕМОЙ ИНФРАСТРУКТУРЫ

Компоненты инфраструктуры не должны изменяться после их создания. Изменения могут вноситься только путем повторного создания компонента (и любых зависимых компонентов) с новыми или измененными свойствами.

Решение, которое мы только что приняли, влечет за собой компромисс: стоимость удаления и воссоздание конфигураций. Поэтому нам нужно принять дополнительные решения, чтобы упростить процесс. В противном случае мы сами будем сопротивляться внесению изменений в инфраструктуру из-за сложности и дороговизны. Первое решение, которое мы примем, уже упоминалось ранее в книге. Создадим нашу платформу в облаке.

КЛЮЧЕВОЕ РЕШЕНИЕ: РЕАЛИЗОВАТЬ ИНФРАСТРУКТУРУ В ОБЛАКЕ

Компоненты инфраструктуры будут развертываться и управляться на облачной платформе.

Это решение о создании нашей инфраструктуры микросервисов в облаке — важный фактор, способствующий внедрению неизменяемой инфраструктуры. Без этого затраты на приобретение физического оборудования, управление серверами и закупку программного обеспечения были бы слишком большими и повлекли бы сложности. Но в облаке компоненты инфраструктуры являются виртуальными. Благодаря виртуальности мы можем обращаться с инфраструктурой как с ПО. Она позволяет создавать и уничтожать серверы и устройства так же просто, как программные компоненты или объекты в объектно-ориентированной системе.

Неизменяемая инфраструктура поможет нам избежать дрейфа конфигураций серверов и улучшит нашу способность по тиражированию и созданию новых сред с состояниями, аналогичными состоянию промышленной среды. Однако нам нужен способ определения инфраструктуры в виде управляемого набора конфигураций. Вот где может помочь принцип IaC.

Инфраструктура как код

Принцип IaC основан на одном сильном ограничении: все изменения инфраструктуры должны быть представлены в виде набора машиночитаемых файлов (или кода). Команды, применяющие это ограничение, могут представить группу файлов, определяющих целевое состояние инфраструктуры, и воссоздать среду, повторно применив код, который ее создал. Управление кодом инфраструктуры превращается в способ управления состоянием инфраструктуры. В конечном счете применение принципа IaC означает, что мы можем регулировать изменения в наших средах, управляя написанием, тестированием и развертыванием инфраструктурного кода.

Принцип IaC очень важен для поддержки неизменяемой инфраструктуры. Неизменность требует, чтобы мы управляли определениями объектов, чтобы их можно было изменять путем воссоздания. Есть много способов сделать это, но IaC позволяет относиться к инфраструктуре, как к приложениям. С помощью IaC создание и изменение компонентов подобны запуску программы. Мы сможем применить к инфраструктуре наши ноу-хау из мира разработки приложений.

IaC хорошо подходит для системы, которую мы пытаемся создать, поэтому оформим это решение с помощью записи ADR в реестре решений.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИНЯТЬ IAC

Все изменения в инфраструктуре должны определяться в управляемых файлах кода. Изменения не должны вноситься вручную людьми-операторами никаким другим способом.

Чтобы использовать подход IaC, нам понадобится инструмент, который позволит определять желаемые изменения в виде машиночитаемых файлов кода. Этот инструмент также должен будет интерпретировать файлы IaC и применять их к целевой среде. Много лет назад нам, возможно, пришлось бы создавать этот инструмент самостоятельно, но сейчас доступно множество средств, которые могут выполнить эту работу за нас. Для нашего проекта мы используем инструмент Terraform от компании HashiCorp, с помощью которого будем определять изменения и применять их к облачной среде.

Введение в Terraform

Terraform — популярный инструмент для команд, которые используют принцип IaC и управляют своей инфраструктурой автоматизированным, воспроизводимым способом. Мы добились успеха, используя Terraform в собственных проектах, и, как показал наш опрос практиков, это популярный выбор многих разработчиков. В данной модели мы решили использовать Terraform в качестве инструмента для изменений инфраструктуры, поэтому начнем с документирования этого решения.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНИТЬ TERRAFORM ДЛЯ ИЗМЕНЕНИЙ В ИНФРАСТРУКТУРЕ

Мы будем использовать инструмент Terraform от HashiCorp для управления и применения изменений к инфраструктуре платформы.

Terraform не единственный инструмент, который может помочь нам с изменениями инфраструктуры, в широком доступе существует множество популярных альтернатив. Решение использовать Terraform обосновано тем, что он использует декларативный подход к управлению инфраструктурой. Итак, мы объявляем целевое состояние для инфраструктуры, а Terraform выполняет сложную работу, чтобы привести инфраструктуру в это состояние. Этим он сильно отличается от традиционных инструментов управления конфигурацией, требующих передать им пошаговые инструкции.

Terraform также использует принцип неизменяемой инфраструктуры, который мы решили принять ранее. На практике это позволяет писать код Terraform, описывающий желаемое состояние компонентов инфраструктуры. Когда мы применяем код, Terraform выполняет сложную работу по удалению старой и воссозданию новой инфраструктуры, включая любые зависимые объекты.

Чтобы разработать план приведения среды в конечное состояние, которое мы определили, Terraform должен отслеживать ее текущее состояние. Этим состоянием необходимо тщательно управлять, и им должны делиться все, кто использует данный инструмент. Эффективное управление решением Terraform означает управление состоянием, файлами конфигурации, а также качеством, безопасностью и ремонтопригодностью всего решения (точно так же, как в случае с программным приложением).

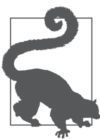
Если вы хотите узнать больше о Terraform, то начать его изучение можно с официальной документации (https://oreil.ly/qaMM5).

Непрерывная интеграция и непрерывное развертывание

Неизменность и IaC делает изменения в нашей инфраструктуре более предсказуемыми. Но такие изменения могут быть небезопасными. Например, что произойдет, если небольшое изменение в настройках сети непреднамеренно приведет к сбою балансировщика нагрузки в рабочей среде? Или если изменение, предназначенное только для среды разработки, случайно просочится в рабочую среду и вызовет сбой?

Один из способов снизить риски — тщательно проверять (и перепроверять) каждое изменение. Но такой подход замедляет скорость внесения изменений из-за большого объема работы по проверке, которую необходимо было бы выполнить. Это также может привести к позднему обнаружению проблем, которые желательно найти как можно раньше в процессе проектирования и разработки инфраструктуры. В итоге на этапе тестирования тратится слишком много времени на устранение большого количества проблем, способных коренным образом изменить наш план инфраструктуры.

Намного эффективнее применять практики DevOps, такие как непрерывная интеграция и непрерывное развертывание (CI/CD). Вместо планирования масштабного тестирования изменений непосредственно перед их переносом в промышленную версию мы будем *непрерывно* интегрировать наши изменения в хранилище, постоянно проверять их работу и автоматически развертывать. Цель состоит в том, чтобы наладить ритмичный выпуск небольших и легко проверяемых изменений.



Понимание CI/CD

Если вы хотите понять принципы CI/CD и научиться эффективно их внедрять, то мы рекомендуем книги Continuous Integration[1] Пола М. Дюваля и Continuous Delivery[2] Джеза Хамбла и Дейвида Фарли (обе — издательства Addison-Wesley).

Практика CI/CD в значительной степени зависит от инструментария. Программные средства позволяют командам эффективно выполнять большие объемы тестов для своего кода. Обычно автоматизация интеграции и тестирования ПО и инфраструктуры требует множества различных инструментов. Вот почему мы будем использовать специальный инструмент, называемый *конвейером* (pipeline). Он позволяет определять этапы процесса CI/CD и управлять ими. Таким образом, любые изменения кода, которые мы вносим, могут автоматически интегрироваться и развертываться каждый раз одним и тем же способом. Формализуем наше решение использовать конвейер CI/CD в этом проекте.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ СИСТЕМНЫЕ ИЗМЕНЕНИЯ С ПОМОЩЬЮ КОНВЕЙЕРА CI/CD

Все изменения должны применяться с помощью автоматизированного конвейера и/или инструмента. Не должно быть никаких изменений, вносимых с помощью инструкций в командной строке или в консоли оператора.

Мы будем использовать конвейер для применения всех изменений, а не только для инфраструктурных. В этой главе мы сосредоточимся на конвейере внесения изменений в нашу инфраструктуру. Позже, в главе 10, мы определим конвейер CI/CD для микросервисов. Существует множество доступных средств реализации конвейеров, поэтому нам нужно принять еще одно решение по выбору инструмента. Для нашей модели мы решили использовать GitHub Actions.

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ GITHUB ACTIONS ДЛЯ КОНВЕЙЕРОВ CI/CD

Команды должны использовать GitHub Actions для реализации конвейеров CI/CD инфраструктуры и микросервисов.

---

[1] *Дюваль П. М., Матиас С. М., Гловер Э.* Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска.

[2] *Хамбл Д., Фарли Д.* Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ.

На момент написания этих строк GitHub Actions был относительно новым продуктом и не настолько богат функциями, как более известные варианты, например Jenkins и GitLab. Мы выбрали GitHub Actions, поскольку планируем использовать GitHub для управления нашим кодом. Возможность использовать единый инструмент для управления кодом и CI/CD заманчива. Это вдвойне верно для данной книги, где мы ограничены рамками печатной страницы.

К концу этой главы мы построим конвейер CI/CD в GitHub Actions, а также настроим конвейер для обработки кода Terraform и внесения изменений в облачную среду. В главе 7 мы используем этот конвейер для подготовки инфраструктуры микросервисов. Но первый шаг, который мы сделаем, — установим некоторые инструменты и настроим рабочую среду.

Настройка среды IaC

Когда вы пишете код приложения, вам нужна среда разработки с инструментами, которые позволяют писать код, управлять им, тестировать и запускать. То же касается кода инфраструктуры. В этом разделе мы настроим две среды — локальную и облачную — и используем их для разработки, тестирования и публикации кода инфраструктуры.

Настройка GitHub

В первую очередь мы должны выбрать способ управления нашим кодом. Мы будем использовать Git с хостингом на GitHub. Существует множество отличных предложений Git-хостинга, и GitLab — один из самых популярных. Для нашей модели мы решили использовать GitHub, поскольку он стал очень популярным местом для обмена кодом. Это полезно для реализации, поскольку мы будем делиться с вами большим количеством кода и конфигураций по мере создания нашего примера приложения.

КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ GITHUB ДЛЯ УПРАВЛЕНИЯ КОДОМ

Весь код будет управляться с помощью системы управления версиями Git и размещаться на GitHub.

Для работы с нашими примерами вам нужно будет создать учетную запись на GitHub и установить локально клиент Git. Git — невероятно популярный инструмент управления исходным кодом, поэтому, скорее всего, он уже установлен на вашем компьютере и вы знаете, как им пользоваться. Если у вас еще не установлен клиент Git, то посетите страницу загрузки Git (https://oreil.ly/5Vlcy) и следуйте инструкциям, чтобы скачать соответствующую версию.

Если вы новичок в Git, то мы рекомендуем вам начать с главы «Основы Git» в книге Pro Git[1] Скотта Чакона и Бена Штрауба (https://oreil.ly/CK29D), которую они любезно сделали доступной бесплатно в Интернете. Можете также посетить Git Handbook (https://oreil.ly/raKSF) от GitHub, если просто ищете краткую информацию о том, что такое Git и почему он полезен.

В дополнение к клиенту Git вам понадобится учетная запись GitHub, чтобы вы могли управлять своим кодом и настраивать собственные конвейеры CI/CD. Если у вас еще нет учетной записи GitHub, то вы можете зарегистрироваться, чтобы получить ее бесплатно (https://oreil.ly/WcXv-).

Мы будем использовать Git и GitHub для управления кодом наших микросервисов. Но, следуя принципу IaC, с помощью этих инструментов мы также будем управлять кодом инфраструктуры. Он будет написан на специальном языке HCL, который Terraform сможет понять. Перейдем к установке клиента Terraform.

Установка Terraform

Как упоминалось ранее, мы будем использовать Terraform для декларативного управления нашей инфраструктурой и применения изменений к ней. Наш план состоит в том, чтобы автоматически запускать клиент Terraform в конвейере CI/CD. Поскольку этот конвейер будет размещен в GitHub, вам на самом деле не нужно устанавливать Terraform на свою рабочую станцию. Однако, как показывает наш опыт, перед передачей кода в конвейер его желательно протестировать локально. Поэтому мы рекомендуем установить Terraform в вашей локальной среде.

На момент написания книги Terraform был доступен для запуска на следующих платформах:

- OS/X;
- FreeBSD;
- Linux;
- OpenBSD;
- Solaris;
- Windows.

[1] *Штрауб Б., Чакон С.* Git для профессионального программиста. — СПб.: Питер, 2015.

Посетите сайт Terraform (https://www.terraform.io), чтобы выбрать и загрузить клиент и установить его на свой компьютер. Мы использовали версию 0.12.20 для всех примеров в этой книге. И оставляем за вами право следовать инструкциям для выбранной вами платформы.

Когда установка будет завершена, выполните следующую команду, чтобы убедиться, что Terraform установился правильно:

```
$ terraform version
```

Вы должны получить примерно такой ответ, в зависимости от установленной версии:

```
Terraform v0.12.20
```

Мы будем использовать Terraform для управления ресурсами инфраструктуры на облачной платформе. В нашей модели эти ресурсы будут размещены в AWS. Рассмотрим, как и почему мы будем использовать AWS и что нужно сделать, чтобы начать работу с этой платформой.

Настройка Amazon Web Services

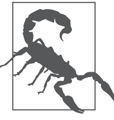
Ранее мы приняли решение использовать облачную инфраструктуру. Но так и не решили, на какой облачной платформе работать. Сегодня существует три облачные платформы, которыми пользуются большинство специалистов по микросервисам: Microsoft Azure, Google Cloud Platform (GCP) и AWS. Мы успешно применяли все в собственных реализациях и даже работали с компаниями, которые использовали все три.

Для нашей модели и примера приложения мы решили воспользоваться услугами одного облачного провайдера. Это упростит и ускорит нашу реализацию. В этой связи мы выбрали AWS прежде всего потому, что на момент написания этой книги у него была самая большая база пользователей. Тем не менее все три перечисленных крупных поставщика облачных услуг предлагают аналогичные сервисы, поэтому вы сможете адаптировать нашу модель для любого из них.

КЛЮЧЕВОЕ РЕШЕНИЕ: РАЗМЕСТИТЬ МИКРОСЕРВИСЫ В AWS

Мы будем использовать AWS в качестве облачной платформы для микросервисов.

Поскольку решено использовать AWS, вам потребуется учетная запись AWS, чтобы следовать нашим примерам. Если у вас ее нет, то вы сможете зарегистрироваться по адресу https://aws.amazon.com. Обратите внимание: для активации вашей учетной записи вам понадобится банковская карта.



Следите за своим счетом

Хотя AWS предлагает бесплатный тариф обслуживания, в примерах книги используются ресурсы, которые предоставляются за плату. Мы дадим вам инструкции по уничтожению любых ресурсов, которые мы создадим, но вы должны будете убедиться, что они уничтожены.

Помимо начальной учетной записи вам необходимо настроить «операционную» учетную запись, чтобы инструменты, которые мы используем, могли получить доступ к вашему экземпляру AWS.

Настройка операционной учетной записи AWS

К концу этой главы мы получим конвейер, способный автоматически развернуть инфраструктуру в AWS. Если мы будем придерживаться принципов «инфраструктура как код» и неизменяемость, то нам никогда не придется управлять инфраструктурой AWS, внося изменения непосредственно через браузер. Но для начала нам нужно выполнить несколько действий вручную, чтобы запустить нашу систему. На первом шаге настроим набор учетных данных и разрешений, позволяющих инструментам работать с нашими объектами AWS.

В AWS управление пользователями, группами и разрешениями осуществляется в рамках сервиса управления идентификацией и доступом (Identity and Access Management, IAM). Нам нужно создать специального пользователя, представляющего наш инструментарий, и определить набор разрешений для наших программных средств. Идентификатор этого пользователя будет передаваться в каждом вызове к нашей конвейерной платформе CI/CD. Как упоминалось ранее, мы будем использовать Terraform в качестве основного инструмента IaC. Следуйте инструкциям ниже, чтобы создать пользователя для Terraform в AWS, который позволит вносить необходимые изменения в среду микросервисов.

Войдите в консоль управления AWS (https://oreil.ly/8LpnE) со своими учетными данными пользователя root. После входа в систему вам должен быть представлен список сервисов AWS. Найдите и выберите сервис IAM — обычно его можно найти в разделе Security, Identity & Compliance (Безопасность, идентификация и соответствие требованиям) (рис. 6.2).

AWS Management Console
AWS services
Find Services
You can enter names, keywords or acronyms.
Example: Relational Database Service, database, RDS
All services
Compute
EC2
Lightsail
Lambda
Batch
Satellite
Ground Station
Quantum Technologies
Amazon Braket
Security, Identity, & Compliance
IAM
Resource Access Manager
Cognito

Рис. 6.2. Выбор IAM

Щелкните на ссылке Users (Пользователи) слева в навигационном меню IAM, а затем нажмите кнопку Add user (Добавить пользователя), чтобы запустить процесс создания пользователя IAM, как показано на рис. 6.3.

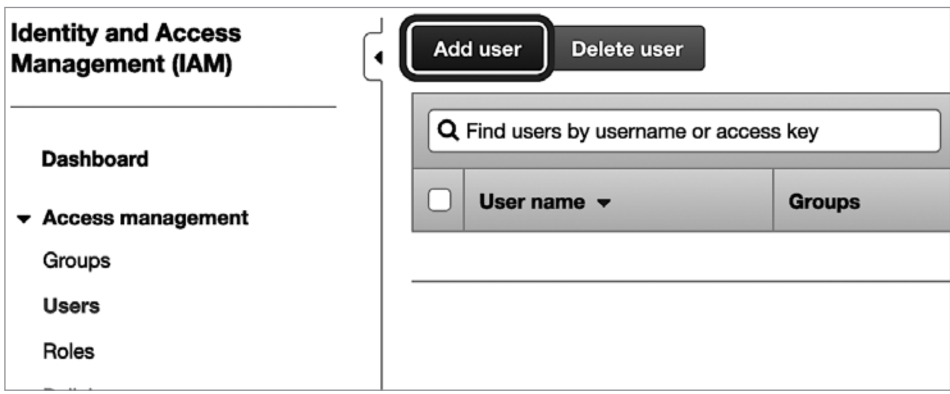


Рис. 6.3. Кнопка Add user (Добавить пользователя)

Введите `ops-account` в поле User name (Имя пользователя). Эту учетную запись мы собираемся использовать также для доступа к CLI и API, поэтому выберите Programmatic access (Программный доступ) в разделе Access type (Тип доступа), как показано на рис. 6.4.

Add user

1 2 3 4 5

Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name* ops-account

Add another user

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type* Programmatic access
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

AWS Management Console access
Enables a **password** that allows users to sign-in to the AWS Management Console.

Рис. 6.4. Ввод данных пользователя

После этого нажмите кнопку Next: Permissions (Далее: Разрешения).

Нашей операционной учетной записи потребуется много разрешений для выполнения работы в AWS от нашего имени. Однако на данный момент мы назначим только один набор разрешений, упакованный в политику AWS под названием `IAMFullAccess`.

Чтобы добавить эту политику, нажмите кнопку Attach existing policies directly (Прикрепить существующие политики напрямую) вверху. Найдите политику `IAMFullAccess` и выберите ее, установив соответствующий флажок, как показано на рис. 6.5.

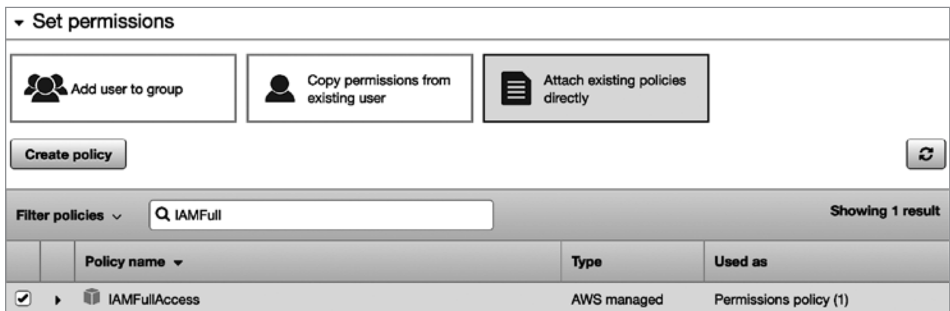


Рис. 6.5. Подключение политики IAMFullAccess

После нажмите кнопку Next: Tags (Далее: Теги). Мы не будем создавать никаких тегов, поэтому просто нажмите кнопку Next: Review (Далее: Обзор) для просмотра сведений о нашем пользователе (рис. 6.6).

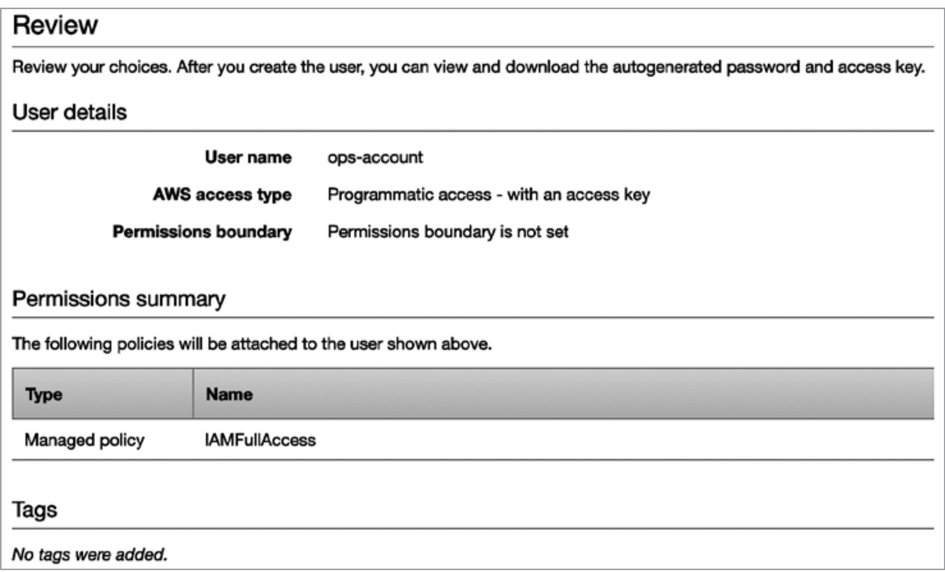
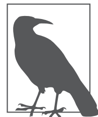


Рис. 6.6. Просмотр сведений о пользователе

Если вас все устраивает, то нажмите кнопку Create user (Создать пользователя). Теперь вы должны увидеть экран, который выглядит примерно так, как показано на рис. 6.7.

Рис. 6.7. Пользователь создан

Прежде чем продолжить, нужно записать ключи вновь созданного пользователя. Щелкните на ссылке Show (Показать), скопируйте значения Access key ID (Идентификатор ключа доступа) и Secret access key (Секретный ключ доступа) во временный файл. Мы используем их оба позже в этом разделе для настройки автоматизированного конвейера. Будьте осторожны с информацией о ключах, поскольку любой завладевший этими ключами сможет создавать ресурсы в вашей среде AWS за ваш счет.



Обязательно запишите где-нибудь оба ключа, прежде чем покинуть этот экран. Они вам понадобятся позже в этой главе.

Мы только что создали пользователя `ops-account`, имеющего разрешение вносить изменения в IAM. Теперь у нас есть все, что нужно, чтобы закрыть консоль управления в браузере и начать использовать приложение AWS CLI, которое мы установили ранее. Первым делом настроим интерфейс командной строки для использования только что созданного операционного пользователя.

Настройка AWS CLI

Облачные провайдеры предлагают три способа управления конфигурациями: браузер, веб-API и интерфейс командной строки (command line interface, CLI). Мы уже использовали браузер для создания операционной учетной записи, а позже будем использовать Terraform для настройки изменений через AWS API. Но нам нужно внести еще некоторые изменения, прежде чем Terraform сможет обращаться к AWS API от нашего имени. Для этого мы прибегнем к AWS CLI.

Использование CLI значительно облегчает описание изменений, которые вам необходимо внести. Он также менее подвержен изменениям, которые претерпевают пользовательские интерфейсы (user interfaces, UI). Но, чтобы использовать CLI, его нужно установить в локальную рабочую среду.

Перейдите на страницу загрузки AWS CLI (https://aws.amazon.com/cli) и следуйте инструкциям, чтобы установить CLI в локальную систему.

Как только все будет готово, первое, что мы сделаем, — настроим CLI, чтобы он мог получить доступ к нашему экземпляру. Выполните команду `aws configure`, как показано в примере 6.1. Вы можете заменить название региона по умолчанию на более близкий вам регион AWS. Полный список регионов AWS доступен на сайте AWS (https://oreil.ly/UrX_t).

Пример 6.1. Настройка AWS CLI

```
$ aws configure
AWS Access Key ID [****************AMCK]: AMIB3IIUDHKPENIBWUVGR
AWS Secret Access Key [****************t+ND]: /xd5QWmsqRsM1Lj4ISUmKoqV7/...
Default region name [None]: eu-west-2
Default output format [None]: json
```

Вы можете проверить правильность настройки интерфейса командной строки, запросив список созданных учетных записей командой `iam list-users`:

```
$ aws iam list-users
{
    "Users": [
        {
            "Path": "/",
            "UserName": "admin",
            "UserId": "AYURIGDYE7PXW3QCYYEWM",
            "Arn": "arn:aws:iam::842218941332:user/admin",
            "CreateDate": "2019-03-21T14:01:03+00:00"
        },
        {
            "Path": "/",
            "UserName": "ops-account",
            "UserId": "AYUR4IGBHKZTE3YVBO2OB",
            "Arn": "arn:aws:iam::842218941332:user/ops-account",
            "CreateDate": "2020-07-06T15:15:31+00:00"
        }
    ]
}
```

Если все сделано правильно, то вы должны увидеть список своих учетных записей в AWS. Он указывает на то, что AWS CLI работает должным образом и имеет доступ к вашему экземпляру.

Теперь можно настроить разрешения, необходимые нашей операционной учетной записи.

Настройка разрешений AWS

Когда мы создали нашего пользователя `ops-account`, то назначили ему политику IAM, которая дает разрешение только на изменение настроек IAM. Но этой учетной записи потребуется гораздо больше разрешений для управления ресурсами AWS и создания нашей инфраструктуры. В этом подразделе мы используем инструмент командной строки AWS и с его помощью создадим и назначим дополнительные политики разрешений операционной учетной записи.

Для начала создадим новую группу `Ops-Accounts` и добавим в нее пользователя `ops-account`. Это позволит нам добавлять новых пользователей в группу, если мы захотим, чтобы у них были те же разрешения. С помощью команды ниже создайте новую группу `Ops-Accounts`:

```
$ aws iam create-group --group-name Ops-Accounts
```

В случае успеха интерфейс командной строки AWS отобразит созданную группу:

```
{
    "Group": {
        "Path": "/",
        "GroupName": "Ops-Accounts",
        "GroupId": "AGPA4IGBHKZTGWGQWW67X",
        "Arn": "arn:aws:iam::842218941332:group/Ops-Accounts",
        "CreateDate": "2020-07-06T15:29:14+00:00"
    }
}
```

Теперь просто добавим нашего пользователя в новую группу, выполнив следующую команду:

```
$ aws iam add-user-to-group --user-name ops-account --group-name Ops-Accounts
```

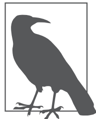
В случае успеха вы не получите ответ от CLI. В данном случае отсутствие новостей — хорошая новость.

Далее нужно назначить набор разрешений нашей группе `Ops-Account`. Эти разрешения будут автоматически применяться к операционным пользователям, добавленным в эту группу. Назначенные нами разрешения позволят пользователю создавать и изменять ресурсы AWS. На практике вам, вероятно, потребуется изменить разрешения для операционного пользователя в процессе проектирования инфраструктуры. В книге мы уже проделали всю работу по проектированию заранее, поэтому точно знаем, какие политики необходимы.

Выполните следующую команду, чтобы назначить все необходимые политики группе `Ops-Accounts`:

```
$ aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/IAMFullAccess &&\
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonEC2FullAccess &&\
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryFullAccess &&\
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonEKSClusterPolicy &&\
```

```
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonEKSServicePolicy &&\
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonVPCFullAccess &&\
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonRoute53FullAccess &&\
aws iam attach-group-policy --group-name Ops-Accounts\
 --policy-arn arn:aws:iam::aws:policy/AmazonS3FullAccess
```



Копия этой команды в виде сценария доступна в репозитории книги на GitHub (https://oreil.ly/Microservices_UpandRunning_scripted).

В дополнение к готовым политикам, которые предоставляет AWS, нам также понадобятся некоторые специальные разрешения для работы с AWS Elastic Kubernetes Service (EKS). Мы подробно познакомимся с EKS в следующей главе, а сейчас нужно разобраться с разрешениями. В AWS нет предопределенной политики с необходимыми разрешениями, которую мы могли бы назначить, поэтому создадим и назначим свою политику.

Для этого создайте файл `custom-eks-policy.json` и заполните его кодом, приведенным в примере 6.2. Этот файл также можно найти в репозитории книги на GitHub (https://oreil.ly/Microservices_UpandRunning_json).

Пример 6.2. Нестандартная политика JSON для EKS

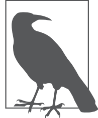
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "eks:DescribeNodegroup",
        "eks:DeleteNodegroup",
        "eks:ListClusters",
        "eks:CreateCluster"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "eks:*",
      "Resource": "arn:aws:eks:*:*:cluster/*"
    }
  ]
}
```

Теперь выполните следующую команду, чтобы создать новую политику `EKS-Management` на основе только что созданного файла JSON:

```
$ aws iam create-policy --policy-name EKS-Management\
 --policy-document file://custom-eks-policy.json
```

В случае успеха эта команда выведет представление новой политики в формате JSON:

```
{
    "Policy": {
        "PolicyName": "EKS-Management",
        "PolicyId": "ANPA4IGBHKZTP3CFK4FAW",
        "Arn": "arn:aws:iam::[some_number]:policy/EKS-Management",
        "Path": "/",
        "DefaultVersionId": "v1",
        "AttachmentCount": 0,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "CreateDate": "2020-07-06T15:50:26+00:00",
        "UpdateDate": "2020-07-06T15:50:26+00:00"
    }
}
```



В AWS каждый ресурс имеет уникальный идентификатор, называемый именем ресурса Amazon (Amazon Resource Name, ARN). Строка цифр в ARN только что созданной политики будет уникальной для вас и вашего экземпляра AWS. Выпишите строку ARN вашей политики, чтобы можно было ссылаться на нее в следующих шагах.

После создания новой политики остается лишь назначить ее нашей группе пользователей. Запустите следующую команду, заменив текст *`{YOUR_POLICY_ARN}`* идентификатором ARN вашей политики:

```
$ aws iam attach-group-policy --group-name Ops-Accounts \
   --policy-arn {YOUR_POLICY_ARN}
```

Теперь у вас есть пользователь `ops-account`, имеющий разрешения для автоматического создания ресурсов инфраструктуры AWS. Мы будем использовать эту учетную запись при написании кода Terraform и настройке конвейера инфраструктуры. Обязательно сохраните секретный ключ и ключ доступа где-нибудь под рукой (и в безопасности), так как они понадобятся позже.

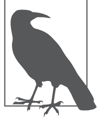
Прежде чем приступить к созданию конвейера, нужно выполнить последнюю настройку: создать корзину (bucket) AWS S3, где Terraform будет хранить состояние.

Создание серверного хранилища S3 для Terraform

Инструмент Terraform весьма эффективен, поскольку позволяет определять желаемое состояние инфраструктуры вместо конкретных шагов для приведения ее в это состояние. Terraform творит чудеса, внося нужные изменения в среду, чтобы она выглядела так, как мы описали. Но для этого инструмент должен отслеживать ее текущее состояние и последние выполненные операции. Terraform сохраняет всю эту информацию в файле JSON, который читает и обновляет при каждом запуске.

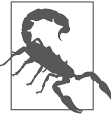
По умолчанию Terraform хранит этот файл в локальной файловой системе. Но на практике хранение файла состояния в локальной системе не лучшее решение. Состояние часто должно совместно использоваться всеми компьютерами и пользователями, чтобы можно было управлять средой из разных мест. Однако к локальным файлам состояния трудно предоставить общий доступ, из-за чего можно легко столкнуться с конфликтами состояний и проблемами синхронизации.

Чтобы избежать этих неприятностей, мы сохраним файл состояния Terraform в сервисе AWS S3. Terraform изначально поддерживает использование S3 в качестве серверного хранилища состояния. От нас требуется лишь создать новую «корзину» для данных и установить правильные разрешения для операционной учетной записи.



Как и большинство облачных провайдеров, AWS предоставляет множество различных вариантов хранения данных. Сервис хранения Amazon Simple Storage Service (S3) позволяет создавать объекты данных, на которые можно ссылаться с помощью ключа. Для Amazon объекты данных — это просто большие двоичные объекты, они могут иметь любой формат, который вам нравится. В этом случае Terraform будет хранить состояние среды в виде объектов JSON.

Чтобы создать корзину, ей нужно присвоить уникальное имя и выбрать регион, в котором она должна находиться. Вы уже должны были выбрать регион по умолчанию при настройке AWS CLI, и мы рекомендуем использовать его же для корзины S3. Более подробную информацию о регионах корзин S3 вы можете найти в документации AWS (https://oreil.ly/5Frfk).

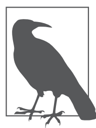


Имена корзин S3 должны быть уникальными

На корзины Amazon S3 можно ссылаться по их именам. Таким образом, выбранное имя должно быть уникальным во всем выбранном вами регионе AWS. Высока вероятность, что вы не сможете использовать распространенные имена вроде test или microservices. Поэтому вам придется придумать что-то уникальное. Обычно достаточно добавить ваше имя к имени корзины. На протяжении всей книги, всякий раз ссылаясь на эту корзину S3, мы будем использовать текст {YOUR_S3_BUCKET_NAME} и оставим вам возможность заменить его вашим именем корзины.

Если вы размещаете свою корзину в регионе us-east-1, то используйте следующую команду:

```
$ aws s3api create-bucket --bucket {YOUR_S3_BUCKET_NAME} \
> --region us-east-1
```



S3 требует особого обращения, если вы не находитесь в регионе us-east-1. В следующих примерах мы перечислили версии команд для региона по умолчанию us-east-1 и остальных. Также для удобочитаемости мы разбили команды на несколько строк с помощью bash-оператора продолжения строки (\).

Если вы размещаете корзину S3 в регионе, *отличном* от us-east-1, то используйте следующую команду:

```
$ aws s3api create-bucket --bucket {YOUR_S3_BUCKET_NAME} \
> --region {YOUR_AWS_REGION} --create-bucket-configuration \
> LocationConstraint={YOUR_AWS_REGION}
```

В случае успеха вы должны увидеть объект JSON с указанием местоположения вашей корзины. Это будет выглядеть примерно так, как в этом примере для корзины `my-msur-test`:

```
{
  "Location": "http://my-msur-test.s3.amazonaws.com/"
}
```

Этот результат подсказывает, что корзина успешно создана и ей был присвоен собственный уникальный URL. По умолчанию корзины S3 не являются общедоступными. И это хорошо, потому что очень нежелательно, чтобы кто-то мог увидеть и изменить наш файл состояния Terraform. Однако мы уже дали операционному пользователю полные права на доступ к сервису S3, поэтому он готов к использованию.

Теперь у нас есть пользователь AWS `ops-account`, наделенный правом создания, редактирования и удаления ресурсов в AWS. Кроме того, ему разрешено хранить объекты в специальной корзине S3, которую мы создали для управления состоянием Terraform. Это был последний раз, когда мы вручную вносили изменения в наш экземпляр AWS. С этого момента мы изменения будут вноситься только с помощью кода и автоматизированного конвейера!

Создание конвейера IaC

Теперь, когда учетные записи, разрешения и инструменты готовы к работе, мы можем перейти к основной теме главы. К концу этого раздела мы создадим готовый к использованию конвейер IaC. Помните, что конвейер инфраструктуры невероятно важен, поскольку дает безопасный и простой способ быстрой подготовки среды. Без конвейера нам пришлось бы вручную выполнить множество шагов для создания сред микросервисов, работающих по-разному.

Вместо этого мы определим стабильное декларативное описание инфраструктуры для наших сервисов. Наши команды разработки и эксплуатации смогут с помощью этого описания создавать собственные среды для тестирования, внесения изменений и выпуска сервисов в работу. В данной главе мы не будем создавать никаких реальных инфраструктур AWS, а только реализуем основу, которой воспользуемся в главе 7. В этом разделе мы создадим следующие компоненты:

- репозиторий Git на GitHub для изолированной тестовой среды — «песочницы»;
- корневой модуль Terraform, определяющий эту «песочницу»;
- конвейер GitHub Actions CI/CD, создающий «песочницу».

«Песочница», которую мы собираемся создать, — это просто тестовая среда, которая даст нам возможность опробовать наши модули и конвейеры IaC. Мы создадим ее в следующей главе, а затем, порадовавшись, что все работает, выбросим ее. Позже мы используем все эти ресурсы и создадим тестовую среду для микросервисов, которые будем проектировать и создавать.

Но нашим первым шагом будет создание хранилища для кода и конвейера, поэтому начнем с создания репозитория.

Создание репозитория «песочницы»

В начале этой главы мы уже упоминали, что будем использовать Git и GitHub для управления нашим кодом инфраструктуры. Если вы следовали указаниям, то у вас уже установлен клиент Git и имеется учетная запись GitHub, готовая к использованию. Мы применим оба этих инструмента и создадим новый репозиторий для «песочницы».

В нашей модели мы решили создать для каждой среды собственный репозиторий с включенным в него кодом и конвейером. Нам нравится этот подход, поскольку он позволяет командам быть более независимыми в управлении средами и при этом хранить вместе конфигурацию конвейера и код для простоты.

> **КЛЮЧЕВОЕ РЕШЕНИЕ: ОТДЕЛЬНЫЙ РЕПОЗИТОРИЙ ДЛЯ КАЖДОЙ СРЕДЫ**
>
> Код и конвейер каждой среды будут храниться независимо в собственном репозитории.

Для создания репозитория «песочницы» мы используем веб-интерфейс GitHub. Создать новый репозиторий можно также с помощью приложения GitHub CLI, но с веб-интерфейсом сделать это будет быстрее и проще. Позже мы снова задействуем веб-интерфейс GitHub для запуска и мониторинга конвейера.

Некоторым практикующим специалистам нравится сохранять все конфигурации среды вместе в одном общем репозитории. Это упрощает совместное использование библиотек, компонентов и действий всеми средами и помогает поддерживать согласованность. Большинство специалистов также используют специализированные инструменты CI/CD (Jenkins — один из самых популярных), а не создают новые внутри GitHub. Это важное решение, поэтому вам нужно будет оценить компромиссы при создании следующей архитектуры микросервисов, базируясь на наблюдениях, которые вы делаете на основе создаваемой вместе с нами системы.

Чтобы создать репозиторий, откройте браузер и перейдите на страницу входа в GitHub (https://github.com/new). Если вы еще не вошли в свою учетную запись GitHub, то вам будет предложено ввести свои учетные данные. После этого вам будет представлена форма для создания нового репозитория. Дайте новому репозиторию имя `env-sandbox` и выберите Private (Закрытый) в параметрах доступа. Также установите флажок Add .gitignore и выберите Terraform в раскрывающемся списке, как показано на рис. 6.8.

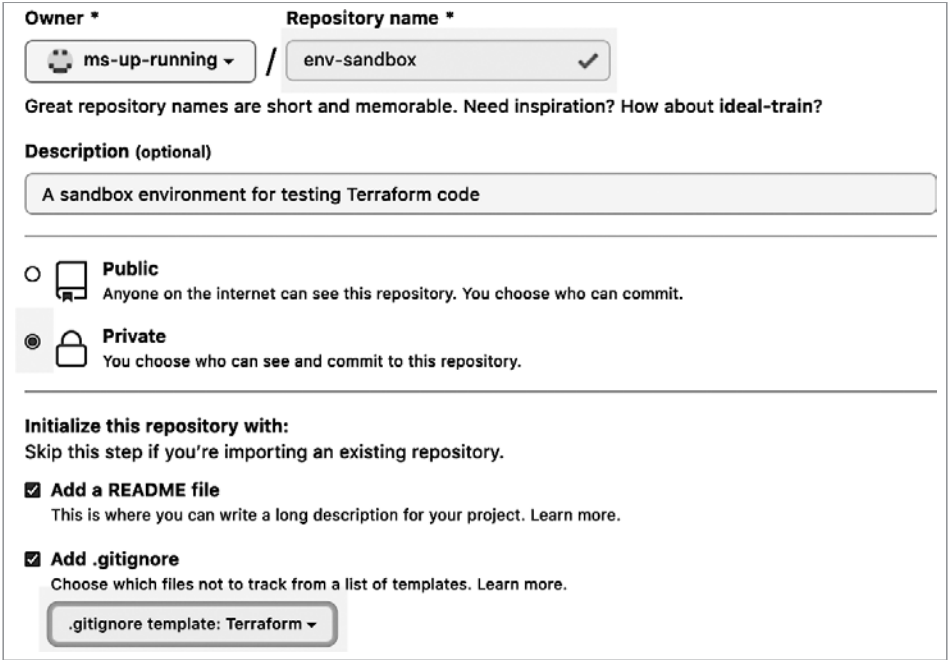


Рис. 6.8. Создание репозитория для «песочницы» в GitHub

Важно не забыть попросить GitHub добавить `.gitignore` для Terraform в модуль, чтобы избежать случайной отправки в репозиторий скрытых рабочих файлов Terraform. Если вы пропустили этот шаг, то сможете добавить файл позже, скопировав исходный код на сайте GitHub (https://oreil.ly/VZ0Xk).

Код можно писать с помощью текстового веб-редактора GitHub, но выполнять реальные работы таким образом не очень удобно. Поэтому мы клонируем репозиторий в локальную среду разработки, чтобы иметь возможность использовать собственные инструменты.

Оставим на ваше усмотрение создание клона репозитория `env-sandbox` в вашей локальной среде разработки.

Если вы никогда раньше не работали с Git и GitHub, то полезные инструкции о том, как клонировать репозиторий GitHub, вы найдете в официальной документации GitHub (https://oreil.ly/tZXaG).

На данный момент это все, что нужно было сделать на GitHub. Мы вернемся к веб-интерфейсу GitHub позже, когда будем работать над конвейером. Но, имея созданный локальный клон, мы можем начать работу над кодом Terraform.

Понимание Terraform

Как упоминалось ранее, мы будем использовать Terraform как основной инструмент для декларативного описания основы нашей инфраструктуры. Terraform выполняет много сложной работы, чтобы внести изменения, соответствующие объявленному состоянию. Тем не менее начать работу с этим инструментом на удивление легко, а язык, который он использует, прост и понятен. Благодаря этому он отлично подходит для определения архитектуры и достижения нашей цели — запустить систему как можно быстрее.

Файлы Terraform записываются в формате HCL, который был изобретен в HashiCorp (компании, создавшей Terraform). HCL похож на JSON, но имеет некоторые улучшения. Если вы привыкли к JSON, то самое большое отличие, которое вы заметите, заключается в использовании другого разделителя в парах «ключ — значение». Вместо двоеточия (`:`) ключи и значения в формате HCL разделяются пробелом или символом `=` в зависимости от контекста.

Есть и некоторые другие незначительные улучшения, такие как комментарии и возможность размещения строк в нескольких строках. По нашему опыту, это простой язык с очень пологой кривой обучения, особенно если вы использовали JSON или YAML в прошлом.

Помимо самого формата HCL полезно изучить четыре ключевые концепции Terraform: хранилища состояний, провайдеры, ресурсы и модули.

- *Хранилище состояния*. Terraform должен хранить файл состояния, чтобы знать, какие изменения необходимо внести в инфраструктуру. Хранилище состояния — это местоположение файла состояния. По умолчанию файл хранится в локальной файловой системе. Мы же будем использовать корзину AWS S3, которую настроили ранее.
- *Ресурсы*. Ресурс — это объект, представляющий предмет, для которого определяется состояние. Terraform вносит изменения, чтобы привести ресурс в это состояние.
- *Провайдеры*. Провайдер Terraform — это упакованная библиотека ресурсов, которую вы можете применить в своем коде. По большей части мы

будем использовать провайдер AWS. Самое приятное, что эта концепция применима к множеству различных облачных платформ и инфраструктур, — вам просто нужно указать провайдер, который вы планируете использовать.

- *Модули*. Модули Terraform подобны функциям или процедурам на обычном языке программирования. Они позволяют инкапсулировать код на HCL в многоразовые модули.

В Terraform намного больше понятий, чем мы описали здесь, но знания этих концепций вполне достаточно, чтобы начать работу по созданию среды. Желающим узнать больше мы рекомендуем для начала обратиться к документации Terraform (https://oreil.ly/07b_c).

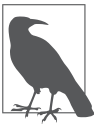
Наш следующий шаг — написать код Terraform, который поможет создать среду «песочницы».

Написание кода для «песочницы»

Наша цель в этой главе — настроить инструментарий и инфраструктуру для сборки среды, поэтому отложим до следующей главы разработку законченного файла Terraform, определяющего инфраструктуру. На данный момент нам нужно создать простой стартовый файл для тестирования инструментов на основе Terraform.

В процессе работы инструмент Terraform CLI отыскивает файлы в текущем рабочем каталоге. В частности, он ищет файл `main.tf`, анализирует его и затем применяет изменения, чтобы воссоздать состояние, описанное в файле. Очевидно, что в любом каталоге может находиться только один файл `main.tf`, поэтому мы выделим отдельный каталог для нашей «песочницы» и создадим в нем файл `main.tf` с описанием ее целевого состояния.

Мы уже создали репозиторий Git `env-sandox` для «песочницы», поэтому будем использовать этот каталог для хранения кода Terraform. Создайте новый файл `main.tf` в локальном Git-репозитории «песочницы». Заполните его кодом на HCL из примера 6.3.



Вам потребуется заменить текст {YOUR_S3_BUCKET_NAME} и {YOUR_AWS_REGION} именем корзины S3, которую вы создали ранее, и выбранным вами регионом AWS.

Пример 6.3. env-sandbox/main.tf

```
terraform {
  backend "s3" {
    bucket = "{YOUR_S3_BUCKET_NAME}"
    key    = "terraform/backend"
    region = "{YOUR_AWS_REGION}"
  }
}

locals {
  env_name         = "sandbox"
  aws_region       = "{YOUR_AWS_REGION}"
  k8s_cluster_name = "ms-cluster"
}

# Конфигурация сети

# Конфигурация EKS

# Конфигурация GitOps
```

Здесь под именем корзины S3 подразумевается простое ее имя (например, my-bucket), а не полный URL.

Только что написанный фрагмент кода на HCL сообщает Terraform, что в роли хранилища состояния мы используем корзину S3. Он также определяет набор локальных переменных с помощью конструкции `locals`. В конце файла присутствует несколько комментариев, отмечающих места, куда мы будем добавлять описание деталей инфраструктуры. Мы используем локальные переменные и заполним остальную часть конфигурации в следующей главе, а пока просто протестируем заготовку нашего файла Terraform.

Написав первый файл с кодом для Terraform, мы готовы попробовать запустить некоторые команды, чтобы убедиться, что все работает как должно. Инструмент Terraform CLI включает множество полезных функций, способствующих повышению качества и безопасности кода вашей инфраструктуры. Его можно использовать для форматирования (или *статического анализа*) написанного вами кода HCL, проверки синтаксиса и пробного прогона, чтобы оценить изменения, которые Terraform будет выполнять с вашим провайдером.

Если вы следовали инструкциям, приведенным ранее в этой главе, то в вашей рабочей среде должна быть доступна локальная копия Terraform. Перейдите

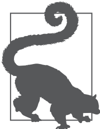
в каталог, где хранится файл `main.tf`, и попробуйте выполнить команду `fmt`, чтобы отформатировать код:

```
env-sandbox msur$ terraform fmt main.tf
```

Команда `fmt` — это средство форматирования. Она проверит ваш файл HCL и внесет изменения, чтобы улучшить согласованность и удобочитаемость кода. Если были внесены какие-либо изменения, то команда выведет имя измененного файла.

Далее мы проверим допустимость синтаксиса написанного нами кода на HCL. Но перед этим нужно установить библиотеки используемого провайдера, в противном случае Terraform сообщит, что не может выполнить проверку синтаксиса. Выполните следующую команду, чтобы установить библиотеки:

```
env-sandbox msur$ terraform init

Successfully configured the backend "s3"! Terraform
will automatically use this backend unless the backend
configuration changes.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform
plan" to see any changes that are required for your infrastructure.
All Terraform commands should now work.

If you ever set or change modules or backend configuration
for Terraform, rerun this command to reinitialize your working
directory. If you forget, other commands will detect it and remind
you to do so if necessary.
```



Если вы получили сообщение об ошибке, связанной с учетными данными AWS, то убедитесь, что выполнили инструкции в начале этой главы и настроили доступ к среде AWS.

Теперь можно запустить команду проверки, чтобы убедиться, что в коде нет никаких синтаксических ошибок:

```
env-sandbox msur$ terraform validate
Success! The configuration is valid.
```

Наконец, можно запустить команду `plan` и посмотреть, какие изменения внесет Terraform, чтобы создать описанную нами среду. Команда выполнит все те же действия, что и при применении кода, но без внесения фактических изменений. Эта команда выполняет пробный прогон, который позволяет

Terraform показать вам свой план по приведению инфраструктуры в требуемое состояние. Используйте следующую команду, чтобы запустить `plan`:

```
$ terraform plan

Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

No changes. Infrastructure is up-to-date.
```

В этом примере Terraform сообщает, что не было обнаружено никаких различий между вашей конфигурацией и реальными физическими ресурсами. А это значит, что никаких действий выполнять не требуется.

Обратите внимание, что команда `plan` вывела малоинтересный результат: `No changes` (Никаких изменений). Это объясняется тем, что мы фактически не определили никаких ресурсов для создания. Зато теперь у нас есть синтаксически верный файл Terraform и мы можем начать создавать «песочницу». Сейчас самое время зафиксировать и отправить файл в репозиторий GitHub, чтобы файл был доступен для использования:

```
$ git add .
$ git commit -m "The sandbox starter file"
$ git push origin
```

Теперь, получив готовый к работе файл Terraform, переключим внимание на конвейер, который будет автоматически использовать этот файл.

Создание конвейера

В этом подразделе мы настроим автоматизированный конвейер CI/CD, который автоматически применит только что созданный файл Terraform. Для этого мы используем встроенный в GitHub инструмент GitHub Actions. Его преимущество заключается в возможности сохранить конфигурацию конвейера рядом с инфраструктурным кодом.

Самый простой способ использовать GitHub Actions — настроить его через веб-интерфейс. Поэтому вернитесь в браузер и перейдите на страницу репозитория «песочницы», созданного ранее в GitHub.

Наш план состоит в том, чтобы добавить ресурсы в учетную запись AWS, созданную ранее в этой главе. Для этого мы должны дать GitHub возмож-

ность использовать ключ доступа и секретный ключ AWS, полученные при создании операционной учетной записи.

Существует множество способов управления секретами в архитектуре микросервисов, но для нашего инструментария DevOps мы просто используем встроенную функцию хранения секретов GitHub.

Настройка секретов

Перейдите в хранилище секретов на GitHub, щелкнув на ссылке Settings (Настройки) на верхней навигационной панели на странице репозитория. Выберите Secrets (Секреты) в меню параметров настройки слева, как показано на рис. 6.9.

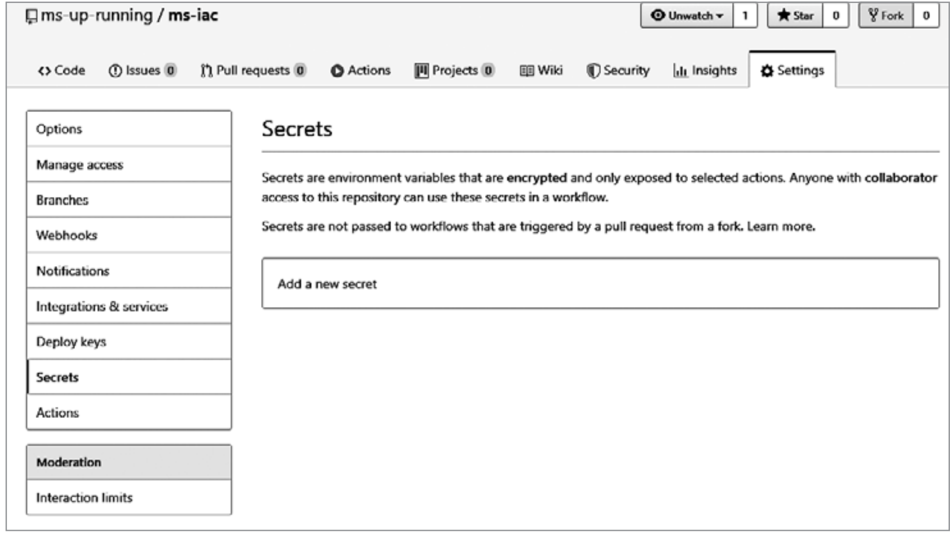


Рис. 6.9. Секреты GitHub

Выберите `Add a new secret` (Добавить новый секрет) и создайте секрет `AWS_ACCESS_KEY_ID`. Введите идентификатор ключа доступа, полученного ранее в этой главе при создании операционного пользователя. Повторите процесс и создайте секрет `AWS_SECRET_ACCESS_KEY` с секретным ключом доступа, тоже полученным ранее. По завершении список секретов должен выглядеть, как показано на рис. 6.10.

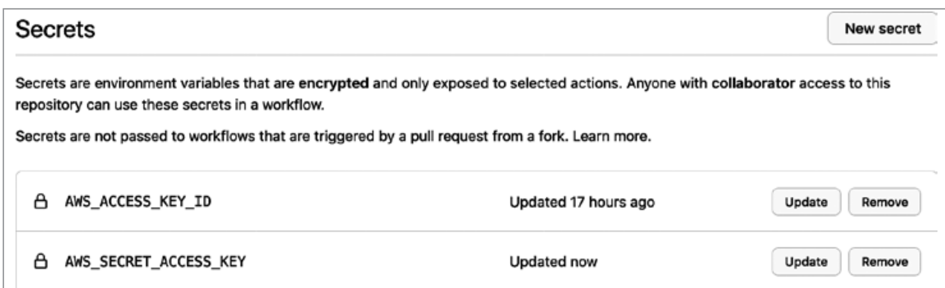


Рис. 6.10. Добавление своего AWS ID и ключа

После добавления секретов можно приступить к созданию конвейера.

Создание потока задач

Поток задач (workflow) — это набор шагов, которые должны выполняться после запуска конвейера. Наш конвейер развертывания инфраструктуры микросервисов должен иметь поток задач, которые проверят файлы Terraform, а затем применят их к нашей «песочнице». Но, кроме тестирования и применения изменений, необходимо выполнить еще несколько шагов до и после применения файлов Terraform.

Для запуска потока задач нужен триггер, который сообщит GitHub, когда это нужно сделать. GitHub Actions предоставляет несколько различных вариантов триггеров, но мы будем использовать механизм `tag` в Git. *Тег* (tag) позволяет присвоить имя или метку определенной точке в истории репозитория Git. Использование тегов в качестве триггера позволяет четко определить версии изменений, которые мы вносим в среду. Этот механизм также дает возможность фиксировать файлы в репозитории без запуска сборки.

После запуска потока задач наш конвейер будет работать с файлами Terraform, которые мы передали в репозиторий. Но нам понадобится выполнить некоторые настройки, чтобы подготовить среду сборки. Сначала нужно установить Terraform и AWS, как мы делали это в нашей локальной среде. Хотя код запускается в GitHub Actions, фактическая сборка происходит на виртуальной машине, поэтому нам также нужно получить копию кода из репозитория.

Наконец, после применения изменений к «песочнице» у нас появится возможность выполнить любые действия по очистке или дополнительной подготовке. В данном случае мы сделаем специальный файл конфигурации доступным для загрузки, чтобы можно было подключаться к среде микросервисов в AWS

с локального компьютера. На рис. 6.11 показано, как будет выглядеть законченный конвейер.

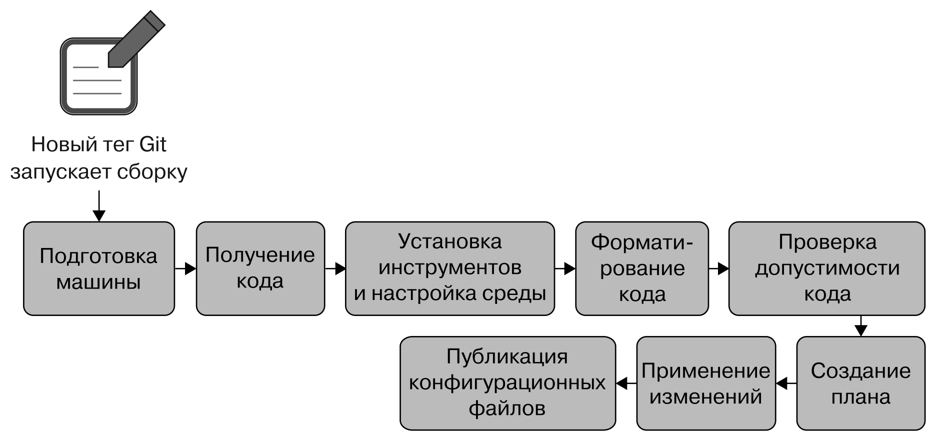


Рис. 6.11. Этапы инфраструктурного конвейера

Для определения этапов конвейера мы используем язык YAML и набор команд потока задач в GitHub Actions. Исчерпывающее их описание вы найдете в документации GitHub Actions (https://oreil.ly/Kk7-J). Углубимся в конфигурацию YAML, перейдя на страницу GitHub Actions в репозитории. Это можно сделать, щелкнув на ссылке Actions (Действия) на верхней навигационной панели на странице репозитория GitHub «песочницы». После перехода вы должны увидеть страницу, как показано на рис. 6.12.

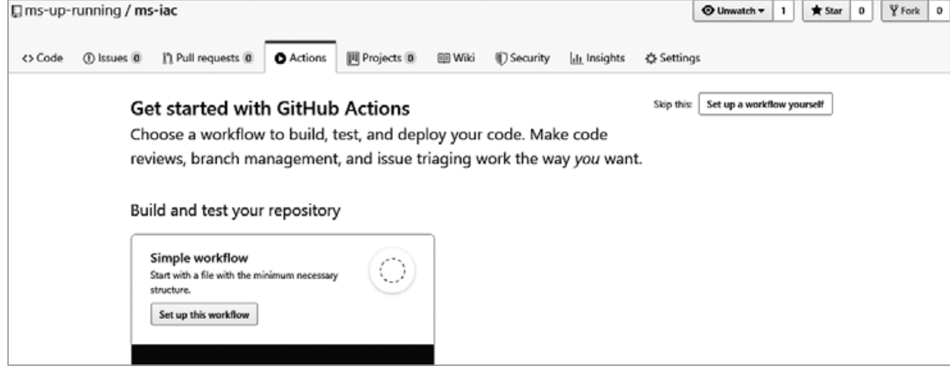
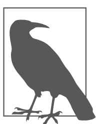


Рис. 6.12. Создание потока задач в GitHub Actions

GitHub Actions предоставляет шаблоны для быстрого создания начального потока задач. Однако мы пойдем другим путем и сами настроим поток задач с нуля. Нажмите кнопку Set up a workflow yourself (Настроить поток задач самостоятельно) вверху справа (или где бы она ни находилась в последней версии интерфейса).

Перед вами откроется веб-редактор с вновь созданным файлом YAML для вашего потока задач. GitHub хранит файлы Actions в скрытом каталоге /.github/workflows. Клонировав репозиторий GitHub, можно редактировать эти файлы в любом редакторе, который вам нравится, или создавать новые файлы YAML для определения новых потоков задач GitHub Actions. Но редактирование в разделе Actions на сайте GitHub дает одно важное преимущество — возможность поиска плагинов на странице Marketplace. Поэтому наш первый поток задач мы будем править в веб-редакторе.

Первым делом настроим триггер для запуска потока задач и контейнерную среду для построения инфраструктуры.



Чтобы вам было проще понять происходящее, мы рассмотрим файл потока задач отдельными частями. Объясним каждую часть по ходу дела, но имейте в виду, что на самом деле поток задач целиком находится в одном файле. Пример законченного файла с потоком задач можно увидеть, перейдя по следующей ссылке на GitHub: https://oreil.ly/Microservices_UpandRunning_env_sandbox.

Настройка и установка триггера

Один из наиболее важных этапов в потоке задач — триггер, инициирующий выполнение потока. Как упоминалось ранее, мы будем использовать простой триггер, основанный на механизме тегов Git. Мы настроим конвейер так, чтобы он запускался всякий раз, когда инфраструктура промаркирована меткой, начинающейся с буквы *v*. Так мы сможем сохранить историю версий созданной нами инфраструктуры. Например, нашу первую сборку инфраструктуры можно пометить как «v1.0».

Для начала замените код YAML в вашем редакторе кодом из примера 6.4.

Пример 6.4. Триггер потока задач и настройка задания

```
name: Sandbox Environment Build

on:
  create:
    tags:
      - v*
```

```
jobs:
  build:
    runs-on: ubuntu-latest
    env:
      AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
      AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

    steps:
    - uses: actions/checkout@v2

    # Установка зависимостей
```

Строка «# Установка зависимостей» в примере 6.4 — это комментарий. Мы будем использовать комментарии YAML для описания происходящего и чтобы показать, где будет добавляться дополнительный код YAML на последующих этапах.

В предыдущем фрагменте `on` — это команда GitHub Actions, которая задает триггер для потока задач. Мы настроили запуск нашего потока задач при создании нового тега, соответствующего шаблону `v*`. Кроме того, мы добавили коллекцию `jobs`, определяющую задания, которые должен выполнить GitHub при ее запуске. Задания должны выполняться на компьютере или в контейнере. В свойстве `runs-on` мы указали, что сборка должна запускаться на виртуальной машине с Ubuntu Linux. Мы также добавили секреты AWS, которые настроили ранее в среде сборки.

Коллекция `steps` определяет конкретные шаги потока задач, которые должны выполняться в настраиваемой среде. Но прежде чем сделать что-нибудь еще, нам нужно получить код. Поэтому первый шаг, который мы определили, получает наш код Terraform из Git с помощью действия GitHub `actions/checkout@v2`. Это действие скопирует в среду сборки Ubuntu код, который будет использоваться остальными шагами задания.

Действия — это модульные библиотеки кода, которые можно вызывать из потока задач GitHub Actions. Действия составляют основу GitHub Actions и обеспечивают богатство возможностей. В магазине GitHub Actions доступно огромное количество действий, которые можно использовать в потоках задач. Но будьте аккуратны при их выборе: создавать и публиковать новые действия может кто угодно, поэтому поддержка, безопасность и качество не гарантированы.

Этот поток задач уже можно запустить, но пока он не делает ничего полезного, кроме получения копии нашего кода. Как вы помните, наша главная цель — начать использовать Terraform, но, чтобы получить такую возможность, нужно

настроить среду для запуска инструментов. Это означает, что мы должны добавить некоторые инструкции по установке зависимостей.

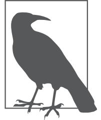
Установка зависимостей

Когда мы настраивали локальную среду разработки инфраструктуры, необходимо было установить инструменты командной строки Git, AWS и Terraform. Нам нужно будет сделать что-то подобное в среде сборки, но, поскольку мы знаем конкретные операции, которые будем выполнять, можно настроить несколько более компактный набор зависимостей.

Хорошая новость в том, что мы получаем Git бесплатно, когда используем GitHub Actions, поэтому не придется беспокоиться о его установке. Кроме того, HashiCorp предоставляет готовое действие GitHub для Terraform, поэтому не стоит переживать об установке клиента Terraform. Единственное, с чем осталось разобраться, — это конфигурация AWS.

Ранее в этой главе мы использовали интерфейс командной строки AWS для внесения изменений в учетную запись AWS. Однако в конвейере они будут вноситься с помощью Terraform. На самом деле мы не собираемся вносить никаких изменений в среду сверх тех, что указали в коде Terraform. Поэтому не потребуется устанавливать AWS CLI.

В итоге нам не нужно устанавливать никаких зависимостей, чтобы получить конвейер, способный создавать для нас ресурсы AWS. Но начав создавать инфраструктуру в следующей главе, мы обнаружим, что ей нужны некоторые дополнительные зависимости для преодоления сложностей с установкой архитектуры микросервисов на основе Kubernetes.



Работая над книгой, мы определили зависимости, которые понадобятся в конвейере еще до того, как вы сами поймете, что они вам нужны. Книги проще читать, когда повествование развивается линейно. Поэтому мы приложили максимум усилий, чтобы дать вам линейную последовательность инструкций. На практике вам придется выполнить несколько итераций редактирования и тестирования действий в конвейере, а также изучить и разработать свои конвейеры развертывания инфраструктуры и микросервисов.

В частности, мы установим инструмент AWS Authenticator и установщик для сервисной сетки Istio. AWS Authenticator — инструмент командной строки, который другие инструменты могут использовать для аутентификации и доступа к среде AWS. Это пригодится позже, когда мы будем работать с Kubernetes и нужно будет настроить доступ к кластеру Kubernetes, размещенному в AWS.

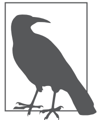
Istio — это инструмент сервисной сетки. Istio будет представлен в следующей главе, а пока просто установим инструменты CLI.

Добавьте код из примера 6.5 в файл потока задач, чтобы настроить эти зависимости в среде сборки. Эти шаги необходимо прописать после комментария `# Установка зависимостей`, который мы добавили ранее. Будьте внимательны при оформлении отступов и убедитесь, что код выровнен с предыдущим шагом `-uses`, поскольку YAML очень привередлив в отношении отступов.

Пример 6.5. Установка зависимостей

```
[...]

    # Установка зависимостей

    - name: Install aws-iam-authenticator
      run: |
        echo Installing aws-iam-authenticator...
        mkdir ~/aws
        curl -o ~/aws/aws-iam-authenticator \
        "https://amazon-eks.s3.us-west-2.amazonaws.com/\
        1.16.8/2020-04-16/bin/linux/amd64/aws-iam-authenticator"
        chmod +x ~/aws/aws-iam-authenticator
        sudo cp ~/aws/aws-iam-authenticator /usr/local/bin/aws-iam-authenticator

  # Применение Terraform
```

Команды `run` в только что добавленном коде YAML будут запускать команды оболочки в среде сборки Ubuntu. Мы добавили инструкции по установке AWS IAM Authenticator, согласно документации AWS, а также инструмента Istio CLI.



Виртуальная машина (VM), определенная в нашем потоке задач GitHub Actions, будет создаваться в момент запуска конвейера и уничтожаться по завершении. Это означает, что инструменты будут устанавливаться каждый раз, когда мы запускаем конвейер, и никакое состояние не будет сохраняться между запусками.

Последняя часть кода YAML использует действие настройки Terraform. Как видите, этот код намного чище и проще для чтения и понимания, чем инструкции установки AWS Authenticator и Istio, выполняемые в командной строке. GitHub Actions особенно хорошо подходит для ситуаций, когда доступны необходимые действия, поэтому мы рекомендуем применять эту платформу, когда она соответствует вашим потребностям.

Теперь, когда наши зависимости настроены и Terraform готов к работе, мы можем добавить в поток задач шаги, в которых используется Terraform.

Применение файлов Terraform

В подразделе «Написание кода для “песочницы”» выше в этой главе мы использовали команды Terraform для форматирования и проверки кода HCL в файле `main.tf`. Нечто подобное нужно сделать и в конвейере, но было бы хорошо, если бы эти действия выполнялись автоматически. Цель состоит в том, чтобы автоматически отформатировать, проверить и спланировать код Terraform. Дополнительно добавим автоматический шаг `apply`, который применит план и внесет изменения.

Добавьте код YAML из примера 6.6 в конец вашего потока задач, после комментария `# Применение Terraform`.

Пример 6.6. Поток задач Terraform

```
# Применение Terraform

      - uses: hashicorp/setup-terraform@v1
    with:
      terraform_version: 0.12.19

  - name: Terraform fmt
    run: terraform fmt

  - name: Terraform Init
    run: terraform init

  - name: Terraform Validate
    run: terraform validate -no-color

  - name: Terraform Plan
    run: terraform plan -no-color

  - name: Terraform Apply
    run: terraform apply -no-color -auto-approve

  # Публикация ресурсов
```

Как видите, мы использовали действие `run` для вызова Terraform CLI из командной строки Ubuntu. Это практически те же действия, что мы выполнили в локальной среде, только на этот раз добавился шаг `apply`, который внесет реальные изменения в инфраструктуру AWS. Обратите внимание на флаг `-auto-approve` в команде `apply`, он избавляет от необходимости диалога с человеком.

Мы почти закончили. Осталось сделать последний шаг — опубликовать любые файлы, полученные в процессе выполнения конвейера.

Публикация ресурсов и внесение изменений

Когда поток задач GitHub Actions завершается, виртуальная машина, использованная для сборки, уничтожается. Но иногда требуется сохранить некоторые данные, файлы или результаты для последующего использования. Для этого GitHub предоставляет действие `upload-artifact`, позволяющее сохранить файлы для последующей загрузки.

В следующей главе мы настроим кластер Kubernetes на AWS. При работе с Kubernetes полезно иметь возможность подключения к кластеру с вашего удаленного компьютера. Для этого потребуется указать много сведений о подключении и аутентификации. Мы упростим эту задачу, представив последний этап, генерирующий файл конфигурации Kubernetes. Его можно скачать, чтобы подключиться к кластеру после его создания.

Добавьте код из примера 6.7 в конец файла с потоком задач, чтобы реализовать этот последний шаг.

Пример 6.7. Загрузка kubeconfig

```
# Публикация ресурсов
- name: Upload kubeconfig file
  uses: actions/upload-artifact@v2
  with:
    name: kubeconfig
    path: kubeconfig
```

Это действие выгружает файл `kubeconfig` из локального рабочего каталога среды сборки в репозиторий GitHub Actions. Предполагается, что такой файл существует, поэтому мы создадим его в следующей главе, когда перейдем к деталям построения нашей инфраструктуры «песочницы».

Теперь у вас есть законченный конвейер для развертывания среды «песочницы». Файл с потоком задач может храниться в GitHub как обычный код. Поэтому зафиксируем изменения, чтобы сохранить их. Нажмите кнопку Start commit (Начать фиксацию), введите описание фиксации и нажмите кнопку Commit new file (Зафиксировать новый файл), чтобы завершить фиксацию изменения (рис. 6.13).

Дальнейшее развитие конвейера

Мы не смогли вместить в эту главу все, что должно быть помещено в конвейер CI/CD IaC для воссоздания промышленной среды. В частности, нам пришлось исключить из конвейера интеграционное тестирование. Поэтому мы настоятельно рекомендуем вам изучить и реализовать этап интеграционного тестирования для кода Terraform. Когда вы начнете внедрять такую функциональность, обратите особое внимание на инструмент Terratest (https://oreil.ly/UKvMQ), написанный на Go в компании Gruntworks.io.

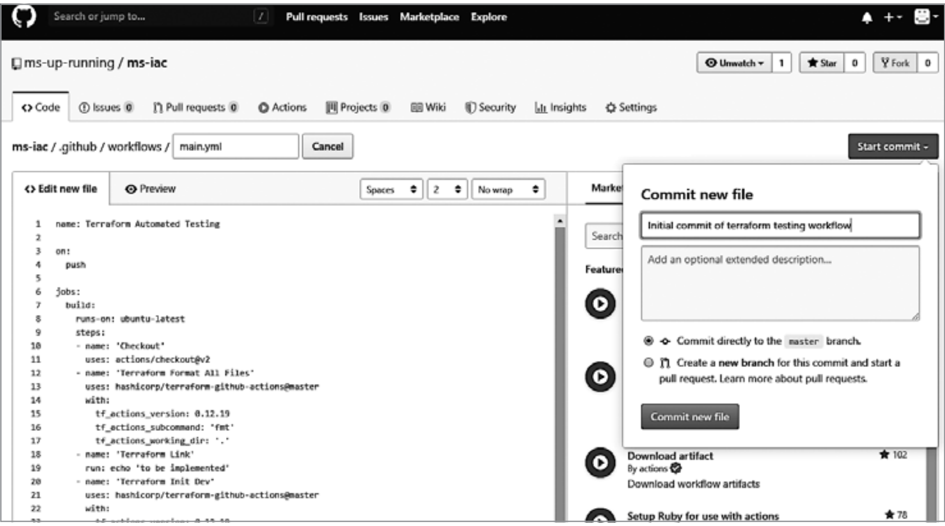


Рис. 6.13. Фиксация изменений в GitHub Actions

Все, что осталось, — опробовать наш поток задач, чтобы убедиться в правильности его работы.

Тестирование конвейера

Чтобы протестировать созданный нами конвейер, нужно активировать триггер, запускающий задание, которое мы определили. В данном случае нужно создать в репозитории тег Git с меткой, начинающейся с буквы *v*. Это можно сделать в веб-интерфейсе, использовав функцию *Releases*. Но, поскольку мы будем выполнять бóльшую часть работы за пределами GitHub на локальной рабочей станции, вместо этого создадим тег там.

Первое, что нужно сделать, — обновить локальный клон репозитория и получить внесенные изменения. Для этого откройте терминал на своей рабочей станции и запустите команду `git pull` в каталоге env-sandbox. Вы должны получить результат, который выглядит примерно так, как показано в примере 6.8. Он сообщает, что в локальный репозиторий загружен новый файл `.github/workflows/main.yml`.

Пример 6.8. Перенос изменений в локальный репозиторий

```
env-sandbox msur$ git pull
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
```

```
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
From https://github.com/msur/env-sandbox
   a6b706f..9923863  master     -> origin/master
Updating a6b706f..9923863
Fast-forward
 .github/workflows/main.yml | 54 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 54 insertions(+)
 create mode 100644 .github/workflows/main.yml
```

Теперь, обновив содержимое репозитория на GitHub, можно создать тег. Поскольку это всего лишь тест, обозначим наш выпуск как v0.1. Используйте команду `git tag`, как показано в примере 6.9, чтобы создать новый тег с меткой.

Пример 6.9. Создание тега v0.1

```
env-sandbox msur$ git tag -a v0.1 -m "workflow test"
```

Хотя мы создали тег, он существует только локально в нашем клоне репозитория на рабочей станции. Чтобы запустить поток задач, нужно поместить этот тег в репозиторий на GitHub. Для этого используйте команду `git push` с именем тега, как показано в примере 6.10.

Пример 6.10. Вставка тега в GitHub

```
env-sandbox testuser$ git push origin v0.1
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 165 bytes | 165.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/mitraman/env-sandbox.git
 * [new tag]         v0.1 -> v0.1
```

Мы будем выполнять эту последовательность добавления и отправки тегов всякий раз, когда понадобится запустить конвейер. Отправка тега должна запустить поток задач, который мы создали в GitHub Actions, поэтому теперь нам осталось лишь убедиться, что он успешно запускается.

Чтобы увидеть статус выполнения, откройте веб-интерфейс GitHub и перейдите на вкладку Actions (Действия), как мы делали раньше. Вы должны увидеть сообщение, как на рис. 6.14. Оно указывает, что задание успешно завершено.

Можно также посмотреть более подробную информацию о выполненном задании. Это часто полезно, если задача выполняется не так, как ожидалось, и необходимо выполнить некоторые действия по устранению неполадок. Чтобы просмотреть сведения, выберите поток задач (у нас он называется `Sandbox Environment Build`), затем выберите задание (в нашем случае оно называется `build`). На экране с подробным описанием вы сможете увидеть, что

происходило на каждом этапе выполнения задания после запуска конвейера (рис. 6.15).

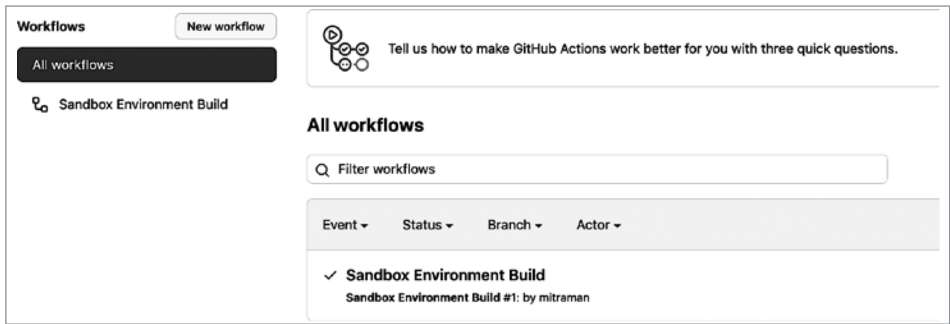


Рис. 6.14. Успешный запуск конвейера

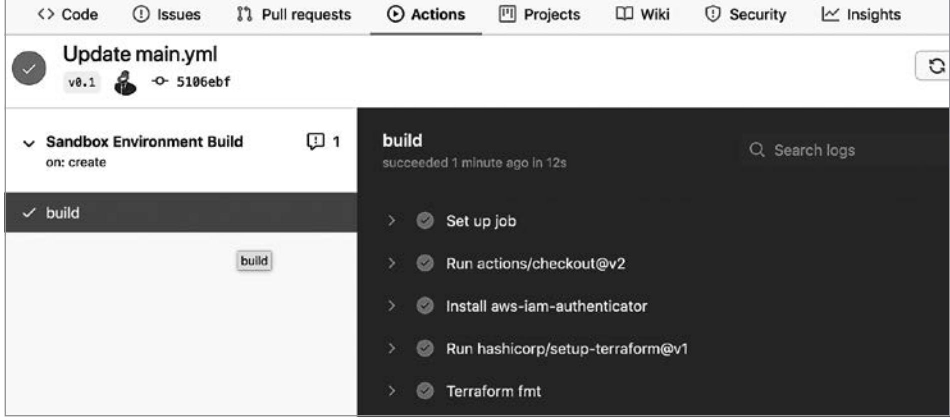


Рис. 6.15. Сведения о выполнении задачи

GitHub Actions — относительно новый продукт, и GitHub часто меняет пользовательский интерфейс, поэтому точная последовательность шагов для перехода к этому экрану может измениться к тому времени, когда вы будете читать этот раздел. Если у вас возникли проблемы с переходом к странице с этапами выполнения задания, то обратитесь к документации GitHub (https://oreil.ly/LPieV).

Успешно протестировав конвейер, мы завершаем настройку инструментов, необходимых для декларативного построения инфраструктуры.

Резюме

В этой главе мы создали простой, но эффективный конвейер IaC, основанный на некоторых важных принципах и практиках DevOps. Установили и использовали Terraform в качестве инструмента для реализации принципов IaC и неизменяемой инфраструктуры. Создали репозиторий на GitHub для управления этим кодом. Наконец, создали поток задач GitHub Actions в виде конвейера CI/CD с автоматическим тестированием, чтобы повысить безопасность и скорость изменений в инфраструктуре.

На самом деле мы не создали никаких инфраструктурных ресурсов, но прошли через этапы внесения изменений, необходимых для этого. Мы создали и отредактировали файл Terraform, протестировали и запустили его локально, зафиксировали в репозитории и добавили тег, чтобы запустить процесс сборки и применения конвейера. Эта последовательность шагов будет нашим методом разработки неизменяемой инфраструктуры, и мы будем часто использовать ее в следующей главе, занимаясь проектированием и созданием инфраструктуры микросервисов.

ГЛАВА 7

Создание инфраструктуры микросервисов

В предыдущей главе мы создали конвейер CI/CD для внесения изменений в инфраструктуру. Инфраструктура для системы микросервисов будет определена в коде, и мы сможем использовать конвейер для автоматизации тестирования и выполнения этого кода. Создав автоматизированный конвейер, можно начинать писать код, определяющий инфраструктуру для приложения на основе микросервисов. Именно на этом мы сосредоточимся в текущей главе.

Настройка правильной инфраструктуры жизненно важна для получения максимальной отдачи от системы микросервисов. Микросервисы дают хорошую возможность разбить приложение на небольшие фрагменты. Но понадобится создать много вспомогательной инфраструктуры, чтобы все эти небольшие сервисы работали слаженно. Однако, прежде чем заняться проектированием и разработкой самих сервисов, нужно уделить некоторое внимание созданию сетевой инфраструктуры и архитектуры развертывания для сервисов.

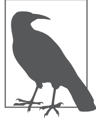
К концу этой главы вы создадите облачную инфраструктуру, предназначенную для размещения микросервисов, которые будут создаваться в следующей главе. Начнем с ознакомления с инфраструктурой и ее компонентами.

Компоненты инфраструктуры

Инфраструктура — это набор компонентов, позволяющих развертывать, сопровождать и поддерживать приложение на основе микросервисов. Она может включать в себя множество частей: аппаратное обеспечение, ПО, сети и инструменты. Таким образом, количество компонентов, которые нужно настроить, довольно велико, и запуск всех этих частей в эксплуатацию — объемная задача.

К счастью, по мере развития подходов к разработке микросервисов было придумано и создано огромное количество инструментов и сервисов, облегчающих эту работу. И мы будем по мере возможности использовать подобные инструменты в нашей модели. Мы также сосредоточимся на реализации инфраструктуры, работающей с единственной облачной платформой (AWS), вместо создания приложения, не зависящего от выбора облачного провайдера, которое можно «поднять и перенести» на другие хосты. Эти решения позволят определить многофункциональную инфраструктуру в ограниченном пространстве одной этой главы.

Но это все равно довольно сложная задача! Нам понадобится рассмотреть множество тем на небольшом количестве страниц. А это означает, что потребуется пойти на некоторые компромиссы. Например, мы не сможем охватить вопросы безопасности, управления операциями, журналирования событий и поддержки. Вместо этого сосредоточимся на разработке и написании кода Terraform для создания действующей сети, управляемого сервиса Kubernetes в AWS и декларативного сервера GitOps. Эти три компонента дадут нам основу для развертывания наших примеров микросервисов.



Сетевое проектирование и Kubernetes — глубокие и сложные темы, требующие намного больше времени на обсуждение, чем мы можем себе позволить. Хорошая новость заключается в том, что вам не нужно быть специалистом по сетям или Kubernetes, чтобы настроить свою первую среду микросервисов. Если это новые сферы для вас, то вы сможете последовать нашим инструкциям и запустить эти части системы как первый шаг к их изучению.

Давайте начнем с краткого обзора основных компонентов и первой обсудим сеть.

Сеть

Микросервисы должны работать в сети. Поэтому нам нужен подходящий вариант. Поскольку мы приняли решение разместить сервисы в облаке AWS, мы должны соответствующим образом адаптировать архитектуру сети и создать виртуальную сеть вместо физической. Мы не будем беспокоиться о деталях физических маршрутизаторов, кабелей или сетевых устройств, а просто научимся использовать язык сетевых ресурсов AWS и соответствующим образом их настраивать.

Мы спроектируем сеть настолько простой, насколько это возможно. Построим ровно столько, сколько необходимо для работы системы. Но нам все равно

нужно будет создать и настроить несколько основных ресурсов для поддержки наших будущих микросервисов.

- *Закрытое виртуальное облако*. В AWS закрытое виртуальное облако (virtual private cloud, VPC) является родительским объектом для виртуальной сети. Мы создадим и настроим VPC как часть нашего сетевой архитектуры.
- *Подсети*. VPC можно разделить на несколько небольших сетей, называемых подсетями. Они позволяют организовать сетевой трафик и контролировать доступ к ресурсам. В нашей сетевой конфигурации мы создадим четыре подсети.
- *Маршрутизация и безопасность*. Помимо VPC и подсетей мы определим объекты, диктующие правила для входящего и исходящего трафика. Например, определим две «закрытые» подсети, которые будут принимать трафик, исходящий только изнутри нашей VPC.

Как можно заметить, в нашей сети есть некоторые сложности, которые нам придется разрешить, включая управление четырьмя «подсетями» и их подключение. Основная причина таких сложностей — требования сервиса Kubernetes. Рассмотрим этот вопрос далее.

Сервис Kubernetes

На протяжении всей книги мы подчеркивали, что снижение затрат на координацию — важный фактор успеха системы. Мы также несколько раз упоминали контейнеры и контейнеризацию в предыдущих главах, потому что контейнеры — отличный способ помочь нашим командам сделать больше с меньшими затратами на координацию. Контейнеры позволяют запускать приложения в предсказуемой и изолированной среде без накладных расходов и сложностей, связанных с развертыванием виртуальных машин. Микросервисы и контейнеры идеально подходят друг другу.

Если вам нужна помощь в понимании контейнеров и контейнеризации, то на сайте Docker есть хорошее вводное объяснение сути контейнеров (https://oreil.ly/tC7aZ).

Контейнеры позволяют легко создавать микросервисы, которые предсказуемо работают в разных средах как автономные единицы. Но контейнеры не знают, как запускаться, масштабироваться или восстанавливать свою работоспособность при поломках. Контейнеры отлично работают сами по себе, но управление

ими в рабочих средах требует немалых усилий. Именно здесь и пригодится Kubernetes.

Kubernetes — инструмент оркестрации контейнеров, разработанный Google. Он решает проблемы управления контейнерами в масштабируемых системах. Kubernetes предоставляет инструментальное решение для развертывания и масштабирования контейнерных приложений, а также наблюдения за ними и управления ими. Он может добавлять и удалять контейнеры, автоматически развертывать или останавливать их в зависимости от уровня нагрузки на систему, монтировать системы хранения, управлять секретами, а также балансировать нагрузку и управлять трафиком. Kubernetes способен решать множество сложных задач, благодаря чему быстро стал неотъемлемой частью стека инфраструктуры микросервисов.

Однако сам Kubernetes тоже довольно сложен. Из-за этого мы не будем вдаваться в подробности его работы. Но мы сможем собрать работающую инфраструктуру Kubernetes, размещенную в AWS.

Желающим узнать больше о Kubernetes мы рекомендуем отличное введение, представленное в книге Kubernetes: Up and Running Брендана Бернса, Джо Беды и Келси Хайтауэра (O'Reilly) (https://learning.oreilly.com/library/view/kubernetes-up-and/9781492046523).

Для тех из вас, кто пока незнаком с Kubernetes, ниже мы перечислили основные компоненты системы, чтобы вы могли следить за тем, как мы ее настраиваем.

- *Кластер Kubernetes* — родительский объект в системе Kubernetes. Устанавливая Kubernetes, вы устанавливаете кластер. Он содержит плоскость управления и набор узлов.
- *Плоскость управления* — «мозг» кластера. Управляет системой, принимая решения о запуске, остановке и репликации контейнеров, а также предоставляет API, который можно использовать для администрирования кластера.
- *Узлы*. Именно они выполняют основную работу. Каждый узел — это физическая или виртуальная машина, на которой выполняется рабочая нагрузка в контейнере. В Kubernetes узлы запускают *модули* (также называемые подами — pod). Каждый модуль содержит один или несколько контейнеров. Каждый кластер имеет по крайней мере один узел.

В нашей реализации мы будем использовать управляемый сервис AWS поддержки Kubernetes под названием Elastic Kubernetes Service (EKS). Мы решили

использовать EKS, потому что он скрывает от нас основные сложности Kubernetes. Он поможет подготовить кластер и даст нам плоскость управления для использования в качестве управляемого сервиса. Нам останется только настроить количество и типы узлов и создать подходящую сеть.

КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ УПРАВЛЯЕМЫЙ СЕРВИС KUBERNETES

Мы будем использовать AWS EKS в качестве управляемого сервиса для нашего кластера Kubernetes.

Последняя часть нашей инфраструктуры — сервер развертывания GitOps. Давайте посмотрим, что это такое и чем он может нам помочь.

Сервер развертывания GitOps

В главе 2 в числе прочих мы упомянули команду выпуска. Она будет отвечать за развертывание микросервисов в рабочей среде. Ожидается, что наши команды по разработке микросервисов будут использовать конвейер CI/CD для интеграции, тестирования, сборки и доставки своих сервисов. Но в нашей операционной модели они не наделены правом развертывания своих сервисов в системе. Мы приняли это решение, потому что, как показывает наш опыт, развертывание в промышленной среде — довольно сложная процедура, требующая особого внимания. Чтобы облегчить работу команды выпуска, связанную с развертыванием, мы используем специальный инструмент: сервер развертывания GitOps.

Непрерывная доставка и непрерывное развертывание

Часть CD в аббревиатуре CI/CD часто становится источником путаницы. Ее интерпретация как непрерывная доставка (Continuous Delivery) или непрерывное развертывание (Continuous Deployment) зависит от того, кого вы спрашиваете. В нашей модели под непрерывной доставкой понимается автоматическая и непрерывная отправка готовых микросервисов в виде контейнеров командами разработчиков, а под развертыванием — отправка этих контейнеров в рабочую среду командой выпуска.

Название *GitOps*, придуманное в компании WeaveWorks, описывает способ работы, где Git используется как «источник истины». Это означает, что все находящееся в Git определяет целевое состояние системы. Как и Terraform, GitOps следует декларативному подходу. Инструменты GitOps должны синхронизировать конфигурации систем, чтобы они выглядели так, как описано в репозитории Git. Они также должны предупреждать операционные группы, если реальный мир отклонился от состояния, определенного в Git.

Argo CD — это инструмент GitOps, упрощающий развертывание приложений в Kubernetes. Мы решили использовать Argo CD для процесса выпуска, поскольку нам нравится декларативный подход GitOps. Без Argo CD нам пришлось бы организовать автоматическое выполнение серии вызовов Kubernetes для развертывания приложения, тогда как при использовании подхода GitOps мы можем просто сообщить инструменту Argo CD, где находится репозиторий Git, и позволить ему выполнять свою работу по поддержанию среды в актуальном состоянии.

Например, настроив Argo CD, мы сможем заставить его просматривать репозиторий с кодом микросервисов. После того, как новые изменения будут зафиксированы и протестированы, Argo CD сможет автоматически развернуть новую версию сервиса. Эта декларативная возможность непрерывного развертывания делает Argo CD отличным продуктом для команд выпуска.

КЛЮЧЕВОЕ РЕШЕНИЕ: РАЗВЕРТЫВАТЬ МИКРОСЕРВИСЫ С ПОМОЩЬЮ ИНСТРУМЕНТА РАЗВЕРТЫВАНИЯ GITOPS

Команды выпуска микросервисов будут использовать Argo CD для управления развертыванием микросервисов в рабочих и производственных средах.

К концу этой главы мы создадим «песочницу» с сервером Argo CD, установленным поверх кластера Kubernetes, управляемого AWS и работающего в сети AWS VPC. Но для этого придется написать много кода Terraform, так что приготовьтесь засучить рукава. Мы углубимся в разработку в следующем разделе.

Создание инфраструктуры

В главе 6 мы решили использовать Terraform для определения нашей инфраструктуры, и GitHub Actions для тестирования и применения изменений в определении. В этом разделе мы разделим наш проект инфраструктуры на отдельные модули Terraform и вызовем их из среды «песочницы», к созданию которой мы приступили в предыдущей главе. Начнем с настройки инструментов, которые понадобятся в рабочем пространстве разработки инфраструктуры.

Установка kubectl

Если вы следовали инструкциям в главе 6, то у вас уже есть среда, готовая к созданию инфраструктуры, включающая:

- экземпляр AWS и настроенную операционную учетную запись;
- инструменты Git, Terraform и AWS CLI, установленные на рабочей станции;
- конвейер GitHub Actions для развертывания инфраструктуры.

Если вы еще не настроили конвейер GitHub Actions или у вас возникли проблемы с его работой, которые мы описывали выше, то скопируйте базовую среду «песочницы», следуя инструкциям в репозитории GitHub этой книги (https://oreil.ly/Microservices_UpandRunning_env_starter).

В дополнение к настройке, которую мы выполнили в предыдущей главе, нужно сделать еще один шаг, чтобы подготовиться к работе в этой главе: установить `kubectl`. После установки сервиса Kubernetes нам понадобится возможность тестирования и взаимодействия с системой Kubernetes. Для этого мы используем приложение командной строки `kubectl`.

Следуйте инструкциям в документации Kubernetes (https://oreil.ly/sji9x), чтобы установить `kubectl`, выбрав вариант, соответствующий вашей операционной системе.

После подготовки рабочего пространства можно переходить к разработке модулей Terraform, определяющих инфраструктуру.

Настройка репозиториев для модулей

Создавая профессиональное программное обеспечение, важно писать чистый, профессионально выполненный код. Когда он слишком сложен для понимания, поддержки или изменения, эксплуатация и сопровождение проекта становятся затратными. Все это справедливо и для нашего инфраструктурного кода.

Применяя подход IaC, мы должны использовать передовые методы работы с кодом в нашем инфраструктурном проекте. К счастью, в нашей отрасли существует множество хороших рекомендаций, помогающих писать простой и понятный код, удобный в сопровождении.

Но есть и плохая новость: не все принципы и практики разработки традиционного программного обеспечения легко реализовать в сфере IaC. Отчасти это связано с тем, что инструментарий и языки для IaC все еще развиваются, а отчасти с тем, что контекст изменения реального физического устройства отличается от модели разработки традиционного ПО.

Тем не менее, Terraform позволяет применить три основные практики программирования, помогающие писать чистый и простой в сопровождении код:

- *модульная организация* — написание небольших функций, отлично выполняющих единственную задачу;
- *инкапсуляция* — сокрытие внутренних структур данных и деталей реализации;
- *предотвращение повторений* — не повторяйтесь (don't repeat yourself, DRY): всякая задача реализуется один раз и в одном месте.

Встроенная в Terraform поддержка определения инфраструктуры в виде модулей поможет нам применить все три метода. Мы сможем определить инфраструктурный код в виде набора повторно используемых инкапсулированных модулей. Далее мы создадим модули для каждой архитектурно значимой части системы: сетей, шлюза API и управляемого сервиса Amazon Kubernetes (EKS). После этого мы реализуем другой набор файлов Terraform, использующих эти модули, — по одному файлу для каждой среды, которую планируется создать, не повторяя одни и те же объявления инфраструктуры в каждом из них (рис. 7.1).

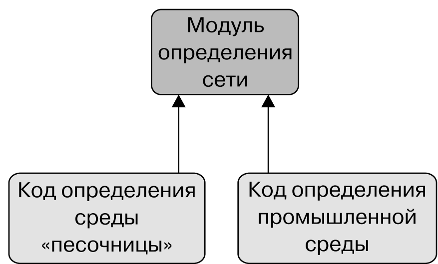
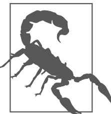


Рис. 7.1. Повторное использование модуля сети

Такой подход позволяет легко развертывать новые среды, создавая новые файлы Terraform, которые повторно используют разработанные модули. Он также дает возможность вносить изменения только в одном месте, чтобы изменить конфигурацию инфраструктуры во всех средах. Мы начнем с создания простого модуля, определяющего базовую сеть, и файла определения среды, использующего этот модуль.

Инфраструктурный код, который мы напишем в этой главе, использует модульную структуру Terraform. Каждый модуль будет храниться в собственном каталоге и содержать файлы `variables.tf`, `main.tf` и `output.tf`. Главное преимущество этого подхода — возможность определить модуль один раз и затем, параметризуя его, создавать несколько сред. Вы можете узнать больше об этих модулях в документации Terraform (https://oreil.ly/87ahC).

Всего мы создадим два модуля для инфраструктуры микросервисов: сетевой модуль AWS с декларативной конфигурацией программно-определяемой сети и модуль Kubernetes, определяющий конфигурацию системы Kubernetes, управляемой AWS. Оба этих модуля мы используем при создании среды «песочницы».



Не используйте наши файлы конфигурации в своей промышленной среде!

Мы постарались спроектировать инфраструктуру, близкую к промышленным средам, которые крупные организации используют для своих микросервисов. Но ограниченность пространства в книге не позволяет нам представить полный обзор конфигураций, которые могли бы соответствовать вашей конкретной среде с точки зрения безопасности и имеющихся у вас ограничений. Вы можете использовать эту главу как начальное руководство и краткий справочник по инструментам, которые вам понадобятся, но мы советуем уделить время разработке собственной производственной инфраструктуры, конфигурации и архитектуры.

В главе 6 мы создали репозиторий GitHub для кода, определяющего среду «песочницы» и ее конвейер CI/CD. В текущей главе мы вновь используем этот репозиторий, но создадим также новый репозиторий для каждого модуля. Terraform имеет встроенную поддержку импорта модулей из репозиториев GitHub, поэтому их будет легко подключить, когда они понадобятся.

Для начала создадим репозитории для всех модулей, которые мы напишем в этой главе. Итак, продолжим и создадим три новых общедоступных репозитория на GitHub, с именами, описанными в табл. 7.1.

Таблица 7.1. Имена инфраструктурных модулей

Имя репозитория	Доступность	Описание
module-aws-network	Общедоступный	Модуль Terraform, создающий сеть
module-aws-kubernetes	Общедоступный	Модуль Terraform, настраивающий сервис EKS
module-argo-cd	Общедоступный	Модуль Terraform, устанавливающий Argo CD в кластер

Если вы не знаете, как создать репозиторий GitHub, то последуйте инструкциям на GitHub (https://oreil.ly/wNY0P).

Рекомендуем сделать эти репозитории общедоступными, чтобы их было легче импортировать в определение среды. При желании вы можете использовать закрытые репозитории, правда, при этом придется добавить информацию для аутентификации в команду импорта, чтобы Terraform мог получить доступ к файлам. Добавьте во вновь созданные репозитории файл `.gitignore`, чтобы исключить отправку большого числа рабочих файлов Terraform на сервер GitHub. Для этого выберите Terraform `.gitignore` в веб-интерфейсе GitHub или сохраните содержимое в виде файла `.gitignore` в корневом каталоге репозитория, как описано на сайте GitHub (https://oreil.ly/7AUJl).

Создав три репозитория GitHub для модулей, можно погрузиться в работу по разработке фактических определений инфраструктуры — начиная с сети.

Модуль определения сети

Виртуальная сеть — основополагающая часть нашей инфраструктуры, поэтому имеет смысл начать с определения сетевого модуля. В этом подразделе мы напишем сетевой модуль AWS, который будет поддерживать определенную архитектуру и рабочую нагрузку Kubernetes и микросервисов. Поскольку это модуль, мы определим в нем код обработки входных данных, реализующий основную логику и возвращающий выходные данные — в точности как в прикладной функции, принимающей входные аргументы, реализующей основную логику и возвращающей результат. Закончив, мы сможем использовать этот модуль для создания сетевой среды, указав всего несколько входных значений.

Код определения сетевой инфраструктуры мы сохраним в репозитории GitHub `module-aws-network`. Мы будем создавать и редактировать файлы Terraform в корневом каталоге этого модуля. Клонируйте репозиторий в локальную систему и подготовьте свой любимый текстовый редактор, если вы этого еще не сделали.

Полный код этого сетевого модуля AWS доступен в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_mod_aws_netw).

Выходные данные сетевого модуля

Начнем с определения ресурсов, которые должен сгенерировать сетевой модуль. Для этого создадим файл Terraform `output.tf` в корневом каталоге module-aws-network, как в примере 7.1.

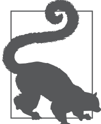
Пример 7.1. module-aws-network/output.tf

```
output "vpc_id" {
  value = aws_vpc.main.id
}

output "subnet_ids" {
  value = [
    aws_subnet.public-subnet-a.id,
    aws_subnet.public-subnet-b.id,
    aws_subnet.private-subnet-a.id,
    aws_subnet.private-subnet-b.id]
}

output "public_subnet_ids" {
  value = [aws_subnet.public-subnet-a.id, aws_subnet.public-subnet-b.id]
}

output "private_subnet_ids" {
  value = [aws_subnet.private-subnet-a.id, aws_subnet.private-subnet-b.id]
}
```

Как можно заметить в этом файле, сетевой модуль создает ресурс VPC, представляющий программно-определяемую сеть для нашей системы. Внутри этой сети модуль также создает четыре логические подсети — это ограниченные разделы нашей сети (или подсети). Две из этих подсетей будут общедоступными, то есть доступными из Интернета. Позже мы используем все четыре подсети в настройках кластера Kubernetes и в конечном итоге развернем наши микросервисы в них.

Конфигурация главного сетевого модуля

Определив выходные данные модуля, можно переходить к декларативному коду, который конструирует ожидаемые выходные данные. Этот код мы поместим в файл `main.tf` в корневом каталоге репозитория `moduleaws-network`.

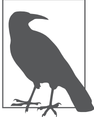


Получение исходного кода

Чтобы помочь вам понять реализацию сети, мы разбили исходный код в файле main.tf на более мелкие части. Полный исходный код этого модуля вы найдете в репозитории книги (https://oreil.ly/Microservices_UpandRunning_maintf).

Начнем реализацию нашего модуля с создания ресурса AWS VPC. Terraform предоставляет специальный ресурс для определения виртуальных машин AWS, поэтому, чтобы создать определение, нам просто нужно заполнить несколько параметров. Создавая ресурс в Terraform, мы определяем параметры и детали конфигурации в синтаксисе Terraform. Когда эти изменения будут применены,

Terraform выполнит вызов AWS API и создаст ресурс, если тот еще не существует.



Вы можете найти всю документацию Terraform для провайдера AWS на сайте Terraform (https://oreil.ly/pvWJS). Вы также можете проконсультироваться с этой документацией, если создаете аналогичную реализацию для GCP или Azure.

Создайте файл `main.tf` в корне репозитория сетевого модуля и добавьте в него код Terraform из примера 7.2 в файл `main.tf`, чтобы определить новый ресурс AWS VPC.

Пример 7.2. modules-aws-network/main.tf

```
provider "aws" {
  region = var.aws_region
}

locals {
  vpc_name = "${var.env_name} ${var.vpc_name}"
  cluster_name = "${var.cluster_name}-${var.env_name}"
}

## Определение AWS VPC
resource "aws_vpc" "main" {
  cidr_block = var.main_vpc_cidr
  tags = {
    "Name"                                        = local.vpc_name,
    "kubernetes.io/cluster/${local.cluster_name}" = "shared",
  }
}
```

Сетевой модуль начинается с объявления используемого *провайдера* AWS. Это специальная инструкция, которая сообщает Terraform, что тот должен загрузить и установить библиотеки, которые понадобятся для взаимодействия с AWS API и создания ресурсов. В процессе проверки или применения этого файла Terraform попытается подключиться к AWS API, используя учетные данные, которые мы настроили в переменных среды. Кроме того, здесь указан регион AWS, чтобы Terraform знал, в каком регионе должен работать.

Мы также определили две локальные переменные, используя блок `locals`. Они определяют стандарт именования, который поможет различать ресурсы среды в консоли AWS. Это особенно важно, если планируется создание нескольких сред в одном пространстве учетных записей AWS, поскольку поможет избежать конфликта имен.

За объявлением локальных переменных следует код создания новой AWS VPC. Как видите, в нем нет ничего особенного, но он определяет две важные вещи: блок CIDR и набор описательных тегов.

Бесклассовая адресация (classless inter-domain routing, CIDR) — стандартный способ описания диапазона IP-адресов сети. Это сокращенная строка, которая определяет допустимые IP-адреса внутри сети или подсети. Например, значение CIDR `10.0.0.0/16` означает, что внутри VPC вы сможете присвоить любой IP-адрес между 10.0.0.0 и 10.0.255.255. Мы определили для среды «песочницы» вполне стандартный диапазон CIDR, но если вам интересно, то за более подробной информацией о том, как работают CIDR и почему они существуют, обращайтесь к документу RFC (https://oreil.ly/PtHmq).

Мы также добавили несколько тегов. Теги ресурсов дают возможность легко идентифицировать группы ресурсов при их администрировании. Также теги могут пригодиться для автоматизации задач и определения ресурсов, которыми следует управлять конкретным образом. В нашем определении мы добавили тег `Name`, чтобы упростить идентификацию VPC. Мы также задали тег Kubernetes, идентифицирующий этот кластер как цель для кластера Kubernetes (который мы определим в пункте «Определение кластера EKS» далее в этой главе).

Кроме того, обратите внимание, что кое-где мы заменили фактическое значение конфигурации ссылкой на переменную. Например, наш блок CIDR определяется как `var.main_vpc_cidr` и имеет тег `Name` со значением `local.vpc_name`. Это переменные Terraform, и мы зададим их значения позже, когда будем использовать данный модуль для определения среды «песочницы». Переменные помогают сделать модули многоразовыми — изменяя значения переменных, можно изменять типы создаваемых сред.

Определив основную VPC, можно переходить к настройке подсетей. Как упоминалось выше в этой главе, для выполнения нашего потока задач мы будем использовать управляемый сервис Amazon Kubernetes (EKS). Для нормальной работы EKS нужно определить подсети в двух разных «зонах доступности». В AWS зона доступности представляет физически отдельный центр обработки данных. Это полезная конструкция: хотя ресурсы AWS являются виртуальными, они все равно работают на компьютере, подключенном к какой-нибудь розетке. Используя две зоны доступности, мы гарантируем, что сервисы будут продолжать работать, даже если один из центров обработки данных выйдет из строя.

В дополнение к настройке двух зон доступности Amazon также рекомендует определять в VPC не только общедоступные, но и закрытые подсети. Поэтому

мы определим в нашей сети общедоступные подсети, разрешающие трафик из Интернета, и закрытые, принимающие трафик только изнутри VPC. При запуске сервис EKS развернет балансировщики нагрузки в общедоступной подсети для управления входящим трафиком, который будет перенаправляться в контейнеры с микросервисами, развернутые в закрытых подсетях.

Для соответствия этим требованиям определим четыре подсети. Две из них будут объявлены общедоступными и потому будут доступны из Интернета. Две другие подсети будут закрытыми. Кроме того, мы развернем общедоступные и закрытые подсети в отдельных зонах доступности. По завершении мы получим сеть, как показано на рис. 7.2.

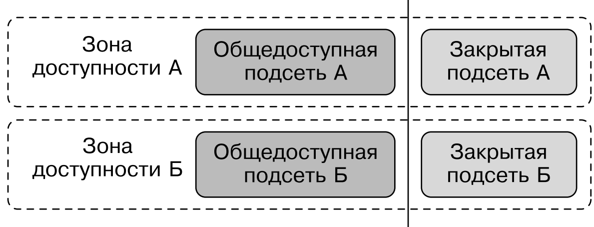


Рис. 7.2. Структура подсетей в AWS

Мы уже определили CIDR с диапазоном IP-адресов для нашей VPC. Теперь нужно распределить эти IP-адреса между подсетями. Поскольку подсети находятся внутри VPC, для каждой нужно определить CIDR в пределах диапазона IP-адресов VPC. Однако мы не будем определять фактические IP-адреса в модуле, а используем переменные, как сделали это для VPC.

В дополнение к блокам CIDR предусмотрим параметры для определения зон доступности подсетей. Для этого используем специальный тип Terraform `data`, позволяющий динамически выбирать название зоны. В данном случае поместим `public-subnet-a` и `private-subnet-a` в `data.aws.availability_zones.available.names[0]`, а `public-subnet-b` и `private-subnet-b` в `data.aws.availability_zones.available.names[1]`. Использование подобных динамических данных облегчает развертывание этой инфраструктуры в разных регионах.

Наконец, добавим тег `Name`, чтобы иметь возможность легко находить наши сетевые ресурсы в консоли администратора и оператора. Добавим также в ресурсы подсетей несколько тегов EKS, чтобы сервис AWS Kubernetes знал, какие подсети мы используем и для чего они предназначены. Отметим общедоступные подсети тегом `elb`, чтобы EKS знал, что может использовать эти

подсети для создания и развертывания балансировщиков нагрузки. Закрытые подсети отметим тегом `internal-elb`, чтобы указать, что в них будут развернуты наши балансируемые рабочие нагрузки. За более подробной информацией об использовании тегов балансировщика нагрузки в AWS EKS обращайтесь к документации AWS (https://oreil.ly/WlqQh).

Добавьте код с определением конфигурации подсети из примера 7.3 в конец файла `main.tf`.

Пример 7.3. modules-aws-network/main.tf (подсети)

```
# Определение подсети

data "aws_availability_zones" "available" {
  state = "available"
}

resource "aws_subnet" "public-subnet-a" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_subnet_a_cidr
  availability_zone = data.aws_availability_zones.available.names[0]

  tags = {
    "Name" = (
      "${local.vpc_name}-public-subnet-a"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb" = "1"
  }
}

resource "aws_subnet" "public-subnet-b" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.public_subnet_b_cidr
  availability_zone = data.aws_availability_zones.available.names[1]

  tags = {
    "Name" = (
      "${local.vpc_name}-public-subnet-b"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/elb" = "1"
  }
}

resource "aws_subnet" "private-subnet-a" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.private_subnet_a_cidr
  availability_zone = data.aws_availability_zones.available.names[0]

  tags = {
```

```
    "Name" = (
      "${local.vpc_name}-private-subnet-a"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb" = "1"
  }
}

resource "aws_subnet" "private-subnet-b" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = var.private_subnet_b_cidr
  availability_zone = data.aws_availability_zones.available.names[1]

  tags = {
    "Name" = (
      "${local.vpc_name}-private-subnet-b"
    )
    "kubernetes.io/cluster/${local.cluster_name}" = "shared"
    "kubernetes.io/role/internal-elb" = "1"
  }
}
```

В Terraform элемент data позволяет запросить информацию у провайдера. В сетевом модуле мы используем элемент данных aws_availability_zones, чтобы запросить у AWS идентификаторы зон доступности в выбранном нами регионе. Это хороший способ избежать жесткого определения значений в модуле.

Мы настроили четыре подсети и их диапазоны IP-адресов, но пока не определили сетевые правила, необходимые AWS для управления трафиком. Чтобы получить законченную конфигурацию сети, добавим таблицы маршрутизации, описывающие допустимые источники трафика для наших подсетей. Например, определим, как трафик будет маршрутизироваться через общедоступные подсети и как подсети будут взаимодействовать друг с другом.

Начнем с определения правил маршрутизации для двух общедоступных подсетей: `public-subnet-a` и `public-subnet-b`. Чтобы сделать эти подсети доступными из Интернета, нужно добавить в VPC специальный ресурс, называемый *интернет-шлюзом*. Это сетевой компонент AWS, соединяющий наше закрытое облако с общедоступной сетью Интернет. Terraform предлагает готовое определение ресурса для шлюза, поэтому используем его и привяжем к нашей VPC с помощью параметра конфигурации `vpc_id`.

После добавления интернет-шлюза определим правила маршрутизации, которые позволят AWS узнать, как пересылать трафик из шлюза в наши подсети. Для этого создадим ресурс `aws_route_table`, который пропускает весь трафик из Интернета (идентифицируемый с помощью блока CIDR 0.0.0.0/0) через шлюз,

а затем просто назначим таблицу маршрутизации нашим двум общедоступным подсетям.

Добавьте код из примера 7.4 в файл `main.tf`, чтобы определить инструкции по маршрутизации для нашей сети.

Пример 7.4. modules-aws-network/main.tf (общественные маршруты)

```
# Интернет-шлюз и таблицы маршрутизации для общедоступных подсетей
resource "aws_internet_gateway" "igw" {
  vpc_id = aws_vpc.main.id

  tags = {
    Name = "${local.vpc_name}-igw"
  }
}

resource "aws_route_table" "public-route" {
  vpc_id = aws_vpc.main.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.igw.id
  }

  tags = {
    "Name" = "${local.vpc_name}-public-route"
  }
}

resource "aws_route_table_association" "public-a-association" {
  subnet_id      = aws_subnet.public-subnet-a.id
  route_table_id = aws_route_table.public-route.id
}

resource "aws_route_table_association" "public-b-association" {
  subnet_id      = aws_subnet.public-subnet-b.id
  route_table_id = aws_route_table.public-route.id
}
```

Определив маршруты для общедоступных подсетей, перейдем к настройке закрытых подсетей. Конфигурация маршрутов для закрытых подсетей немного сложнее, потому что нужно определить маршрут из закрытой подсети в Интернет, чтобы модули Kubernetes могли взаимодействовать с сервисом EKS.

Чтобы такой маршрут работал, для узлов в закрытых подсетях следует определить путь взаимодействия с интернет-шлюзами, которые мы развернули в общедоступных подсетях. Для этого в AWS нужно создать ресурс шлюза,

преобразующего сетевые адреса (Network address translation, NAT), который проложит путь наружу. Ему нужно назначить специальный IP-адрес, называемый elastic IP address (EIP). Это реальный IP-адрес, принадлежащий AWS и доступный из Интернета, в отличие от всех других адресов в нашей сети, которые являются виртуальными и существуют только внутри AWS. Поскольку количество реальных IP-адресов небесконечно, AWS ограничивает их использование. К сожалению, мы не можем реализовать NAT без такого адреса, поэтому нам придется использовать два: по одному для каждого нашего ресурса NAT.

Добавьте в файл `main.tf` код из примера 7.5, чтобы реализовать шлюз NAT в сети.

Пример 7.5. modules-aws-network/main.tf (шлюз NAT)

```
resource "aws_eip" "nat-a" {
  vpc = true
  tags = {
    "Name" = "${local.vpc_name}-NAT-a"
  }
}

resource "aws_eip" "nat-b" {
  vpc = true
  tags = {
    "Name" = "${local.vpc_name}-NAT-b"
  }
}

resource "aws_nat_gateway" "nat-gw-a" {
  allocation_id = aws_eip.nat-a.id
  subnet_id     = aws_subnet.public-subnet-a.id
  depends_on    = [aws_internet_gateway.igw]

  tags = {
    "Name" = "${local.vpc_name}-NAT-gw-a"
  }
}

resource "aws_nat_gateway" "nat-gw-b" {
  allocation_id = aws_eip.nat-b.id
  subnet_id     = aws_subnet.public-subnet-b.id
  depends_on    = [aws_internet_gateway.igw]

  tags = {
    "Name" = "${local.vpc_name}-NAT-gw-b"
  }
}
```

В дополнение к созданному шлюзу NAT нужно определить маршруты для закрытых подсетей. Добавьте код из примера 7.6 в файл `main.tf`, чтобы завершить определение сетевых маршрутов.

Пример 7.6. modules/network/main.tf (закрытые маршруты)

```
resource "aws_route_table" "private-route-a" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat-gw-a.id
  }

  tags = {
    "Name" = "${local.vpc_name}-private-route-a"
  }
}

resource "aws_route_table" "private-route-b" {
  vpc_id = aws_vpc.main.id
  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat-gw-b.id
  }

  tags = {
    "Name" = "${local.vpc_name}-private-route-b"
  }
}

resource "aws_route_table_association" "private-a-association" {
  subnet_id      = aws_subnet.private-subnet-a.id
  route_table_id = aws_route_table.private-route-a.id
}

resource "aws_route_table_association" "private-b-association" {
  subnet_id      = aws_subnet.private-subnet-b.id
  route_table_id = aws_route_table.private-route-b.id
}
```

Вот и все, что требуется для определения основной логики развертывания сети. Запустив этот файл Terraform, мы получим программно-определяемую сеть AWS, готовую для размещения Kubernetes и микросервисов. Но перед этим нам нужно еще определить входные переменные, используемые модулем. Terraform требует, чтобы все входные переменные, на которые ссылается модуль, были определены в файле `variables.tf`. Если этого не сделать, то мы не сможем передавать значения переменных в модуль.

Переменные сетевого модуля

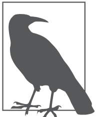
Создайте в корневой папке сетевого модуля файл `variables.tf`. Добавьте в него код из примера 7.7, чтобы определить входные данные для модуля.

Пример 7.7. modules/network/variables.tf

```
variable "env_name" {
  type = string
}

variable "aws_region" {
  type = string
}

variable "vpc_name" {
  type    = string
  default = "ms-up-running"
}

variable "main_vpc_cidr" {
  type = string
}

variable "public_subnet_a_cidr" {
  type = string
}

variable "public_subnet_b_cidr" {
  type = string
}

variable "private_subnet_a_cidr" {
  type = string
}

variable "private_subnet_b_cidr" {
  type = string
}

variable "cluster_name" {
  type = string
}
```

Как видите, определения переменных достаточно понятны. Они описывают имена и типы значений. В нашем модуле мы используем только строковые значения (`string`). В некоторых случаях мы также задали значения по умолчанию, чтобы избавиться от необходимости определять входные данные для каждой

среды. Мы передадим модулю значения для этих переменных, когда будем использовать его для создания среды.



Рекомендуется включать в определения переменных атрибут description с описанием. Это упростит сопровождение и повысит удобство использования ваших модулей, особенно с течением времени. Мы добавили такие описания в файлы Terraform, опубликованные в GitHub, но удалили из примеров в книге, чтобы сэкономить место.

Разработка кода Terraform для сетевого модуля завершена. На данном этапе список файлов в вашем каталоге модуля должен выглядеть примерно так:

```
drwxr-xr-x   3 msur  staff   96 14 Jun 09:57 ..
drwxr-xr-x   7 msur  staff  224 14 Jun 09:58 .
-rw-r--r--   1 msur  staff   23 14 Jun 09:57 README.md
drwxr-xr-x  13 msur  staff  416 14 Jun 09:57 .git
-rw-r--r--   1 msur  staff    0 14 Jun 09:58 main.tf
-rw-r--r--   1 msur  staff  612 14 Jun 09:58 variables.tf
-rw-r--r--   1 msur  staff   72 14 Jun 09:58 outputs.tf
```

Далее мы протестируем сетевой модуль, попробовав создать сетевую среду «песочницы», но прежде проверим — не было ли допущено каких-либо синтаксических ошибок. Приложение командной строки Terraform включает несколько удобных функций для форматирования и проверки кода. Если вы еще не установили клиент Terraform, то сделайте это сейчас, загрузив двоичный файл для вашей операционной системы с сайта Terraform (https://oreil.ly/pFDq8).

Выполните следующую команду Terraform в каталоге модуля, чтобы отформатировать код:

```
module-aws-network$ terraform fmt
```

Команда `fmt` выполнит *статический анализ* и отформатирует весь код Terraform в рабочем каталоге в соответствии со встроенными правилами оформления. Она автоматически внесет эти изменения и отобразит список всех файлов, которые были изменены.

Затем выполните команду `terraform init`, чтобы Terraform мог установить библиотеки провайдера AWS, необходимые для проверки синтаксиса кода. Обратите внимание, что эта команда требует наличия учетной записи на AWS. Если вы еще не создали ее, то последуйте инструкциям, изложенным в предыдущей главе.

```
module-aws-network$ terraform init
```

Если возникнут какие-либо проблемы, то постарайтесь устранить их, прежде чем продолжить. В документации Terraform есть полезный раздел по устранению неполадок (https://oreil.ly/oh_Wn). Наконец, можно запустить команду `validate`, чтобы убедиться, что наш модуль синтаксически корректен:

```
env-sandbox msur$ terraform validate Success!
Success! The configuration is valid.
```

Если вам требуется провести отладку кода Terraform, то определите переменную среды TF_LOG со значением INFO или DEBUG. В этом случае Terraform будет выводить отладочную информацию в стандартный вывод.

Отформатировав и проверив код, изменения в нем можно зафиксировать в репозитории GitHub. Если вы вносили изменения в локальной копии репозитория, то используйте следующую команду, чтобы перенести их в основной репозиторий:

```
module-aws-network$ git add .
module-aws-network$ git commit -m "network module created"
[master ddb7e41] network module created
 3 files changed, 226 insertions(+)
module-aws-network$ git push
```

Теперь наш сетевой модуль на базе Terraform завершен и доступен для использования. В нем есть файл `variables.tf`, описывающий необходимые и необязательные входные переменные, файл `main.tf`, декларативно определяющий ресурсы сети, и, наконец, файл `outputs.tf`, определяющий ресурсы, создаваемые модулем. Теперь попробуем использовать модуль для создания реальной сети в среде «песочницы».

Создание сети «песочницы»

Самое приятное в использовании модулей Terraform заключается в возможности легко воссоздавать среды. Помимо конкретных значений, которые определены в файле `variables.tf`, любая среда, создаваемая с помощью определенного нами модуля, будет работать с сетевой инфраструктурой, которую мы знаем и понимаем. Это означает, что можно ожидать предсказуемое поведение микросервисов по мере их прохождения через среды тестирования и выпуска, поскольку уровень вариативности был снижен.

Но, чтобы применить модуль и создать новую среду, нужно вызвать его из файла Terraform, который определяет значения для переменных модуля. Для демонстрации практического применения модуля Terraform мы создадим среду «песочницы». Если вы выполнили действия, описанные в главе 6, то у вас уже есть репозиторий кода для среды «песочницы», содержащий один файл `main.tf`.

Чтобы применить созданный нами сетевой модуль, задействуем специальный ресурс Terraform `module`. Он позволяет сослаться на модуль Terraform и передать значения для определенных нами переменных. Terraform ожидает обнаружить в ресурсе `module` свойство `source`, указывающее, где найти код.

В нашем случае требуется, чтобы Terraform извлек сетевой модуль из репозитория GitHub. Соответственно, мы должны определить свойство `source`, значение которого начинается со строки `"github.com"` и содержит путь к репозиторию. Это поможет Terraform понять, что он должен извлечь исходный код из GitHub.

Например, значение `"github.com/implementing-microservices/moduleaws-network"` ссылается на наш пример сетевого модуля. Путь к репозиторию своего модуля вы можете скопировать из URL на GitHub (табл. 7.2).

Таблица 7.2. Сетевая переменная среды «песочницы»

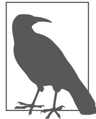
Название	Описание	Пример
YOUR_NETWORK_ MODULE_REPO_PATH	Путь к репозиторию вашего модуля на GitHub	github.com/implementingmicroservices/ module-aws-network

Определив путь к сетевому модулю, откройте файл `main.tf` в репозитории среды «песочницы», который вы создали в главе 6. Добавьте в него код из примера 7.8 после комментария `# Конфигурация сети`. Не забудьте заменить значение `source` на путь к репозиторию GitHub вашего сетевого модуля.

Пример 7.8. env-sandbox/main.tf (сеть)

```
...
# Конфигурация сети
module "aws-network" {
  source = "github.com/{YOUR_NETWORK_MODULE_REPO_PATH}"

  env_name              = local.env_name
  vpc_name              = "msur-VPC"
  cluster_name          = local.k8s_cluster_name
```

```
  aws_region            = local.aws_region
  main_vpc_cidr         = "10.10.0.0/16"
  public_subnet_a_cidr  = "10.10.0.0/18"
  public_subnet_b_cidr  = "10.10.64.0/18"
  private_subnet_a_cidr = "10.10.128.0/18"
  private_subnet_b_cidr = "10.10.192.0/18"
}

# Конфигурация EKS

# Конфигурация GitOps
```



Имена корзин Amazon S3 должны быть глобально уникальными, поэтому замените значение bucket на что-то уникальное и значимое для вас. Инструкции по настройке серверной части находятся в подразделе «Создание серверного хранилища S3 для Terraform» главы 6. Для проверки на скорую руку опустите определение серверного хранилища, и Terraform сохранит состояние локально в вашей файловой системе.

Инфраструктурный конвейер Terraform автоматически применяет изменения, но прежде чем запустить его, стоит убедиться, что написанный нами код Terraform будет работать. Хороший первый шаг — локальное форматирование и проверка синтаксиса кода:

```
$ terraform fmt
[...]
$ terraform init
[...]
$ terraform validate
Success! The configuration is valid.
```

Если в процессе отладки сетевого модуля вы внесли изменения в код, то выполните следующую команду в каталоге среды «песочницы»:

```
$ terraform get -update
```

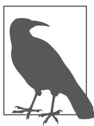
Она даст Terraform указание извлечь последнюю версию модуля из GitHub.

Если ошибок не обнаружится, то можно получить план для оценки изменений, которые Terraform внесет в случае применения плана. Всегда полезно выполнить пробный прогон и изучить планируемые изменения, прежде чем вы фактически измените среду, поэтому сделайте этот этап частью вашего потока задач. Чтобы получить план Terraform, выполните эту команду:

```
$ terraform plan
```

Terraform создаст список ресурсов, которые будут созданы, удалены и обновлены. Если Terraform и AWS для вас в новинку, то вам, возможно, трудно будет оценить и понять план в деталях. Но вы все равно не лишайте себя возможности получить общее представление о том, что должно произойти. Поскольку это первое обновление, в плане должно быть перечислено много новых ресурсов, которые создаст Terraform. Когда вы будете готовы, отправьте код в репозиторий GitHub и создайте `tag`, чтобы произвести выпуск:

```
$ git add .
$ git commit -m "initial network release"
$ git push origin
$ git tag -a v1.0 -m "network build"
$ git push origin v1.0
```



Нам необходимо выполнить две команды git push. Первая фиксирует изменения в коде, а вторая создает тег.

После добавления тега конвейер GitHub Actions должен запуститься и начать создание сети для «песочницы». Чтобы убедиться, что все идет по плану, перейдите на вкладку Actions (Действия) в репозитории среды «песочницы». Если вы не помните, как это сделать, то найдите инструкции в главе 6.

Убедиться в успехе создания VPC можно, выполнив вызов AWS CLI. Запустите следующую команду, чтобы получить список сетей VPC с блоком CIDR, соответствующим тому, который мы определили:

```
$ aws ec2 describe-vpcs --filters Name=cidr,Values=10.10.0.0/16
```

Она должна вывести ответ в формате JSON, описывающий созданную VPC. Если так и произошло, то это означает, что у вас есть готовая к использованию сеть AWS. Теперь пришло время приступить к написанию модуля для сервиса Kubernetes.

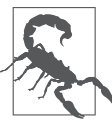
Модуль Kubernetes

Одна из наиболее важных частей нашей инфраструктуры микросервисов — слой Kubernetes, выполняющий оркестрацию наших контейнерных сервисов. При правильной настройке Kubernetes предоставляет автоматизированное решение, обеспечивающее эластичность, масштабирование и отказоустойчивость. Вдобавок мы получим отличную основу для надежного развертывания

своих сервисов. Кроме того, сервисная сетка Istio дает нам эффективный способ управления трафиком и улучшения взаимодействия наших микросервисов.

Чтобы создать свой модуль Kubernetes, выполним те же действия, что и при создании сетевого модуля. Начнем с набора выходных переменных, определяющих ресурсы, создаваемые модулем, затем напишем декларативный код Terraform, применяющий конфигурацию, и, наконец, определим входные данные. Как мы договорились выше в этой главе, каждый модуль инфраструктуры будет храниться в собственном репозитории GitHub. Поэтому прежде создайте новый репозиторий GitHub для модуля Kubernetes, если вы еще этого не сделали.

Внедрение Kubernetes может оказаться очень сложной задачей. Поэтому, чтобы запустить систему как можно быстрее, используем управляемый сервис, который скроет для нас некоторые сложности настройки и управления. В наших примерах мы работаем на AWS, поэтому будем использовать сервис EKS, входящий в состав облачного предложения Amazon.



Конфигурация управляемых сервисов Kubernetes, как правило, зависит от конкретного поставщика, поэтому приведенные здесь примеры, вероятно, потребуют некоторой переработки, если вы решите использовать их в Google Cloud, Azure или другом облачном предложении в Глобальной сети.

Кластер EKS состоит из двух частей: плоскости управления с системным программным обеспечением Kubernetes и группы узлов с виртуальными машинами, где выполняются наши микросервисы. Чтобы настроить EKS, нужно указать параметры для обеих этих частей. Когда модуль завершит работу, можно вернуть идентификатор кластера EKS, чтобы была возможность проверить содержимое кластера или добавить в него что-то еще с помощью других модулей.

Учитывая все вышесказанное, реализуем это решение в коде. Мы будем работать в репозитории GitHub `module-aws-kubernetes`, созданном ранее, поэтому начните с его клонирования на свой локальный компьютер. После этого можно начать редактировать файл Terraform с выходными данными.

Полный листинг модуля Kubernetes доступен в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_Kmod).

Выходные данные модуля Kubernetes

Начнем с объявления выходных данных нашего модуля. Создайте файл Terraform `outputs.tf` в корневом каталоге репозитория `module-aws-kubernets` и добавьте в него код из примера 7.9.

Пример 7.9. module-aws-kubernetes/outputs.tf

```
output "eks_cluster_id" {
  value = aws_eks_cluster.ms-up-running.id
}

output "eks_cluster_name" {
  value = aws_eks_cluster.ms-up-running.name
}

output "eks_cluster_certificate_data" {
  value = aws_eks_cluster.ms-up-running.certificate_authority.0.data
}

output "eks_cluster_endpoint" {
  value = aws_eks_cluster.ms-up-running.endpoint
}

output "eks_cluster_nodegroup_id" {
  value = aws_eks_node_group.ms-node-group.id
}
```

Основное возвращаемое значение — это идентификатор кластера EKS, созданного модулем. Остальные возвращаемые значения необходимы для получения доступа к кластеру из других модулей, как только кластер будет готов и начнет функционировать. Например, нам понадобятся конечная точка и сертификат, когда мы установим сервер Argo CD в этот кластер EKS в конце главы.

Несмотря на простоту выходных данных нашего модуля, настроить систему Kubernetes на базе EKS будет немного сложнее. Как и раньше, создадим файл `main.tf` модуля Terraform по частям, прежде чем протестировать его и применить.

Определение кластера EKS

Для начала создайте файл Terraform `main.tf` в корневом каталоге модуля Kubernetes и добавьте определение провайдера AWS, как в примере 7.10.

Пример 7.10. module-aws-kubernetes/main.tf

```
provider "aws" {
  region = var.aws_region
}
```

Напомним, что мы договорились использовать соглашение `var` об именовании значений, которые могут быть заменены переменными при вызове нашего модуля.

Как упоминалось ранее, мы собираемся использовать сервис EKS от Amazon для создания установки Kubernetes и управления ею. Но для запуска EKS потребуется создавать и изменять ресурсы AWS от нашего имени. Поэтому следует настроить разрешения в учетной записи AWS, чтобы сервис мог выполнять необходимые задачи. Нам нужно определить политики и правила безопасности на уровне кластера, а также на уровне виртуальных машин или узлов, которые EKS будет разворачивать для запуска микросервисов.

Для начала сосредоточимся на правилах и политиках для всего кластера EKS. Добавьте код из примера 7.11 в файл `main.tf`, чтобы определить новую политику управления доступом к кластеру.

Пример 7.11. module-aws-kubernetes/main.tf (управление доступом к кластеру)

```
locals {
  cluster_name = "${var.cluster_name}-${var.env_name}"
}

resource "aws_iam_role" "ms-cluster" {
  name = local.cluster_name

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "eks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
}
```

```
resource "aws_iam_role_policy_attachment" "ms-cluster-AmazonEKSClusterPolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
  role       = aws_iam_role.ms-cluster.name
}
```

Приведенный здесь фрагмент устанавливает политику доверия, которая позволяет сервису AWS EKS действовать от вашего имени. Он определяет новую роль управления идентификацией и доступом для сервиса EKS и к ней политику `AmazonEKSClusterPolicy`. Она определена в AWS и предоставляет EKS разрешения, необходимые для создания виртуальных машин и внесения изменений в сеть в рамках работы по управлению Kubernetes. Обратите внимание, что мы также определяем и используем локальную переменную с именем кластера. И будем использовать ее во всем модуле.

Теперь, когда роль и политика кластерного сервиса определены, добавьте код из примера 7.12 в файл модуля `main.tf`, чтобы задать политику сетевой безопасности кластера.

Пример 7.12. module-aws-kubernetes/main.tf (политика безопасности сети)

```
resource "aws_security_group" "ms-cluster" {
  name        = local.cluster
  vpc_id      = var.vpc_id

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "ms-up-running"
  }
}
```

Группа безопасности VPC ограничивает тип входящего и исходящего трафика. Только что написанный код Terraform определяет правило `egress`, пропускающее исходящий трафик без ограничений, но из-за отсутствия правила `ingress` любой входящий трафик будет запрещен. Обратите внимание, что эта группа безопасности применяется к VPC, которая будет определяться входной переменной. Используя данный модуль, мы можем присвоить ей идентификатор VPC, созданный нашим сетевым модулем.

Определив политики и группы безопасности для кластера EKS, мы можем добавить объявление самого кластера в файл Terraform `main.tf` (пример 7.13).

Пример 7.13. module-aws-kubernetes/main.tf (определение кластера)

```
resource "aws_eks_cluster" "ms-up-running" {
  name     = local.cluster_name
  role_arn = aws_iam_role.ms-cluster.arn

  vpc_config {
    security_group_ids = [aws_security_group.ms-cluster.id]
    subnet_ids         = var.cluster_subnet_ids
  }

  depends_on = [
    aws_iam_role_policy_attachment.ms-cluster-AmazonEKSClusterPolicy
  ]
}
```

Наш кластер EKS имеет довольно простое определение. Оно ссылается на значения имени, роли, политики и группы безопасности, определенные ранее, а также на набор подсетей, которыми будет управлять кластер. Эти подсети мы создали ранее в сетевом модуле, и их можно передать в данный модуль Kubernetes в виде переменной.

Когда AWS создает кластер EKS, автоматически выполняется настройка всех компонентов управления, необходимых для запуска кластера Kubernetes. Все вместе эти компоненты составляют *плоскость управления* — мозг системы Kubernetes.

Но кроме плоскости управления микросервисам нужно место, где они могут работать. Для этого в Kubernetes необходимо настроить узлы — физические или виртуальные машины, на которых могут выполняться контейнерные рабочие нагрузки.

Одним из преимуществ использования управляемого сервиса Kubernetes, такого как EKS, является возможность снять с себя часть обязанностей по управлению созданием, удалением и обновлением узлов Kubernetes. Определим управляемую группу узлов EKS для нашей конфигурации и позволим AWS автоматически предоставлять ресурсы и взаимодействовать с системой Kubernetes. Но, чтобы запустить управляемую группу узлов, нам нужно определить несколько конфигурационных значений.

Определение группы узлов EKS

Начнем настройку узла с определения роли и некоторых политик безопасности точно так же, как проделали с кластером. Добавьте определения IAM группы узлов из примера 7.14 в файл `main.tf` модуля Kubernetes.

Пример 7.14. module-aws-kubernetes/main.tf (группа узлов IAM)

```
# Роль узла
resource "aws_iam_role" "ms-node" {
  name = "${local.cluster_name}.node"

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
}

# Политика узла
resource "aws_iam_role_policy_attachment"
  "ms-node-AmazonEKSWorkerNodePolicy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
  role       = aws_iam_role.ms-node.name
}

resource "aws_iam_role_policy_attachment" "ms-node-AmazonEKS_CNI_Policy" {
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
  role       = aws_iam_role.ms-node.name
}

[...]
resource "aws_iam_role_policy_attachment"
  "ms-node-ContainerRegistryReadOnly" {
[...]
  policy_arn = "arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly"
  role       = aws_iam_role.ms-node.name
}
```

Роль и политики в данном фрагменте кода Terraform позволят любым созданным узлам взаимодействовать с реестрами контейнеров Amazon и сервисами виртуальных машин. Нам нужны эти политики, поскольку узлы в системе Kubernetes должны иметь возможность предоставлять вычислительные ресурсы и получать доступ к контейнерам для запуска сервисов. За более подробной информацией о роли IAM для рабочих узлов EKS обращайтесь к документации AWS EKS (https://oreil.ly/fm75j).

Теперь, определив роль и политики ресурсов узлов, напишем определение для группы узлов, которая их использует. В EKS группе управляемых узлов необходимо указать типы вычислительных ресурсов и ресурсов хранения, которые она будет использовать, а также некоторые ограничения на количество отдельных узлов или виртуальных машин, создаваемых автоматически. Это важно, поскольку мы позволяем EKS автоматически создавать и масштабировать наши узлы. Не хотелось бы непреднамеренно потребить огромное количество ресурсов AWS и в итоге получить внушительный счет.

Можно было бы жестко определить все эти параметры в модуле, но вместо этого мы используем входные переменные со значениями, ограничивающими размер, дисковое пространство и типы процессоров. Благодаря этому с помощью одного и того же модуля Kubernetes мы сможем создавать различные типы сред. Например, среду разработки можно настроить на использование минимальных ресурсов, в то время как для производственной среды выделить больше ресурсов и сделать ее более надежной.

Добавьте код из примера 7.15 в конец файла `main.tf` модуля, чтобы определить группы узлов EKS.

Пример 7.15. module-aws-kubernetes/main.tf (группа узлов)

```
resource "aws_eks_node_group" "ms-node-group" {
  cluster_name    = aws_eks_cluster.ms-up-running.name
  node_group_name = "microservices"
  node_role_arn   = aws_iam_role.ms-node.arn
  subnet_ids      = var.nodegroup_subnet_ids

  scaling_config {
    desired_size = var.nodegroup_desired_size
    max_size     = var.nodegroup_max_size
    min_size     = var.nodegroup_min_size
  }

  disk_size      = var.nodegroup_disk_size
  instance_types = var.nodegroup_instance_types

  depends_on = [
    aws_iam_role_policy_attachment.ms-node-AmazonEKSWorkerNodePolicy,
    aws_iam_role_policy_attachment.ms-node-AmazonEKS_CNI_Policy,
    aws_iam_role_policy_attachment.ms-node-AmazonEC2ContainerRegistryReadOnly,
  ]
}
```

Объявление группы узлов — последняя часть конфигурации EKS. Теперь мы можем попробовать вызвать этот модуль из среды «песочницы» и создать экземпляр кластера Kubernetes в сервисе AWS EKS. Выходные данные модуля вернут

значения, необходимые для подключения к группе узлов после его запуска. Эти сведения о подключении полезно также добавить в файл конфигурации утилиты командной строки `kubectl`, которую большинство операторов используют для управления Kubernetes.

Наш последний шаг — сгенерировать файл `kubeconfig`, который можно использовать для подключения к кластеру. Добавьте код из примера 7.16 в файле `main.tf` модуля.

Пример 7.16. module-aws-kubernetes/main.tf (создать kubeconfig)

```
# Создание файла kubeconfig на основе имеющегося кластера
resource "local_file" "kubeconfig" {
  content  = <<KUBECONFIG_END
apiVersion: v1
clusters:
- cluster:
    "certificate-authority-data: >
   ${aws_eks_cluster.ms-up-running.certificate_authority.0.data}"
    server: ${aws_eks_cluster.ms-up-running.endpoint}
  name: ${aws_eks_cluster.ms-up-running.arn}
contexts:
- context:
    cluster: ${aws_eks_cluster.ms-up-running.arn}
    user: ${aws_eks_cluster.ms-up-running.arn}
  name: ${aws_eks_cluster.ms-up-running.arn}
current-context: ${aws_eks_cluster.ms-up-running.arn}
kind: Config
preferences: {}
users:
- name: ${aws_eks_cluster.ms-up-running.arn}
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1alpha1
      command: aws-iam-authenticator
      args:
        - "token"
        - "-i"
        - "${aws_eks_cluster.ms-up-running.name}"
    KUBECONFIG_END
  filename = "kubeconfig"
}
```

Этот код выглядит сложным, однако на самом деле он довольно прост. Мы используем специальный ресурс Terraform под названием `local_file` для создания файла `kubeconfig`. Затем мы заполняем этот файл содержимым YAML, определяющим параметры подключения к кластеру Kubernetes. Обратите внимание, что значения для файла YAML мы получаем из созданных в модуле ресурсов EKS.

Выполняя этот блок кода, Terraform создаст в локальном каталоге файл `kubeconfig`. Мы сможем использовать данный файл для подключения к среде Kubernetes из инструментов CLI. Мы предусмотрели заполнение `kubeconfig`, когда строили конвейер в главе 6. Запустив конвейер инфраструктуры, вы сможете скачать этот заполненный файл конфигурации и использовать его для подключения к кластеру со своего компьютера.

Написание сервисного модуля Kubernetes почти закончено, осталось только определить переменные.

Переменные модуля Kubernetes

Чтобы объявить переменные для модуля Kubernetes, создайте файл `variables.tf` в репозитории `module-aws-kubernetes` и добавьте в него код из примера 7.17.

Пример 7.17. module-aws-kubernetes/variables.tf

```
variable "aws_region" {
  type        = string
  default     = "eu-west-2"
}
variable "env_name" {
  type = string
}
variable "cluster_name" {
  type = string
}
variable "ms_namespace" {
  type    = string
  default = "microservices"
}
variable "vpc_id" {
  type = string
}
variable "cluster_subnet_ids" {
  type = list(string)
}
variable "nodegroup_subnet_ids" {
  type = list(string)
}
variable "nodegroup_desired_size" {
  type    = number
  default = 1
}
variable "nodegroup_min_size" {
  type    = number
  default = 1
}
```

```
variable "nodegroup_max_size" {
  type    = number
  default = 5
}
variable "nodegroup_disk_size" {
  type = string
}
  variable "nodegroup_instance_types" {
  type = list(string)
}
```

Модуль AWS Kubernetes теперь полностью готов. Как и в случае с сетевым модулем, воспользуемся моментом, чтобы отформатировать и проверить синтаксис кода, выполнив следующие команды Terraform:

```
module-aws-kubernetes$ terraform fmt
[...]
module-aws-kubernetes$ terraform init
[...]
module-aws-kubernetes$ terraform validate
Success! The configuration is valid.
```

Убедившись, что код действителен, зафиксируйте изменения и отправьте их в GitHub, чтобы можно было использовать этот модуль в среде «песочницы»:

```
$ git add .
$ git commit -m "kubernetes module complete"
$ git push origin
```

Закончив с модулем EKS, вернемся к файлу Terraform «песочницы» и используем его.

Создание кластера Kubernetes для «песочницы»

Теперь, заключив сложную систему Kubernetes в простой модуль, мы с легкостью сможем настроить среду «песочницы». Для этого нужно лишь вызвать модуль с нужными входными параметрами. Помните, что наша среда «песочницы» определена в собственном репозитории и имеет свой файл Terraform `main.tf`, который мы использовали для настройки сети. Снова откройте этот файл, но на сей раз добавьте вызов модуля Terraform.

Если вы помните, мы присвоили некоторым из наших входных переменных значения по умолчанию. Для простоты мы используем эти значения в среде «песочницы». Нам также нужно передать некоторые выходные переменные из сетевого модуля в модуль Kubernetes, чтобы развернуть кластер в только что

созданной сети. Но помимо этих входных данных нужно еще определить значение aws_region для процесса установки. Оно должно совпадать со значением, использованным в сетевом модуле и конфигурации серверной части. Также нужно установить параметр source с адресом модуля в GitHub.

Обновите файл main.tf среды «песочницы» и добавьте в него вызов только что созданного модуля Kubernetes. Вы можете добавить ссылку на модуль сразу после комментария # Конфигурация EKS. Замените также текст **{YOUR_EKS_MODULE_PATH}** путем к репозиторию GitHub модуля (пример 7.18).

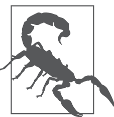
Пример 7.18. env-sandbox/main.tf (Kubernetes)

```
...

# Конфигурация сети
...

# Конфигурация EKS
module "aws-eks" {
  source = "*github.com/{YOUR_EKS_MODULE_PATH}*"

  ms_namespace       = "microservices"
  env_name           = local.env_name
  aws_region         = local.aws_region
  cluster_name       = local.k8s_cluster_name
  vpc_id             = module.aws-network.vpc_id
  cluster_subnet_ids = module.aws-network.subnet_ids

  nodegroup_subnet_ids     = module.aws-network.private_subnet_ids
  nodegroup_disk_size      = "20"
  nodegroup_instance_types = ["t3.medium"]
  nodegroup_desired_size   = 1
  nodegroup_min_size       = 1
  nodegroup_max_size       = 3
}

# Конфигурация GitOps
```

Теперь можно зафиксировать и отправить этот файл в конвейер CI/CD инфраструктуры и создать работающий кластер EKS. Не забывайте, что для запуска сборки нужно создать и отправить тег. Например, вы можете выполнить следующие команды для создания версии инфраструктуры 1.1:

```
$ git add .
$ git commit -m "initial k8s release"
$ git push
$ git tag -a v1.1 -m "k8s build"
$ git push origin v1.1
```

Будьте готовы подождать результата в течение какого-то времени, поскольку подготовка нового кластера EKS может занять от 10 до 15 минут. По завершении вы получите мощную и отказоустойчивую контейнерную инфраструктуру, готовую к запуску ваших микросервисов.



Кластер AWS EKS, который мы определили здесь, будет требовать плату, даже когда простаивает. Мы рекомендуем уничтожать среду, когда вы ее не используете. Инструкции по выполнению этой операции вы найдете в подразделе «Очистка инфраструктуры» далее в главе.

Убедиться в готовности кластера можно, выполнив следующую команду AWS CLI:

```
$ aws eks list-clusters
```

В случае успеха вы получите такой ответ:

```
{
    "clusters": [
        "ms-cluster-sandbox"
    ]
}
```

Наш последний шаг — установить средство развертывания GitOps, которое пригодится, когда придет время запускать сервисы в кластере Kubernetes нашей среды.

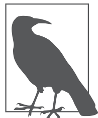
Настройка Argo CD

Как упоминалось ранее, мы собираемся завершить настройку инфраструктуры с помощью сервера GitOps, который будем использовать позже в книге. Мы продолжим следовать выбранному шаблону и создадим модуль Terraform для Argo CD, который будет вызваться для запуска сервера в среде «песочницы». В отличие от других модулей, Argo CD будет устанавливаться в только что созданную систему Kubernetes.

Для этого необходимо сообщить Terraform, что должен использоваться другой хост. До сих пор мы использовали провайдер AWS, который позволяет Terraform взаимодействовать с AWS через свой API. Для установки Argo CD мы используем провайдер Kubernetes. Это позволит Terraform запускать команды Kubernetes для установки приложения в наш новый кластер. Также для установки мы используем систему управления пакетами Helm. Эту систему мы представим немного позже, а пока настроим Terraform, чтобы задействовать его в качестве провайдера.

Этот ресурс будет установлен в кластер Kubernetes, а не на платформу AWS.

Это означает, что вместо провайдера AWS Terraform будет использовать провайдеров Kubernetes и Helm.



Законченная версия данного модуля доступна в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_argo_mod).

Создайте файл `main.tf` в корневом каталоге Git-репозитория `module-argo-cd`, который вы создали ранее. Добавьте код из примера 7.19, чтобы настроить необходимые для установки провайдеры.

Пример 7.19. module-argo-cd/main.tf

```
provider "kubernetes" {
  load_config_file       = false
  cluster_ca_certificate = base64decode(var.kubernetes_cluster_cert_data)
  host                   = var.kubernetes_cluster_endpoint
  exec {
    api_version = "client.authentication.k8s.io/v1alpha1"
    command     = "aws-iam-authenticator"
    args        = ["token", "-i", "${var.kubernetes_cluster_name}"]
  }
}

provider "helm" {
  kubernetes {
    load_config_file       = false
    cluster_ca_certificate = base64decode(var.kubernetes_cluster_cert_data)
    host                   = var.kubernetes_cluster_endpoint
    exec {
      api_version = "client.authentication.k8s.io/v1alpha1"
      command     = "aws-iam-authenticator"
      args        = ["token", "-i", "${var.kubernetes_cluster_name}"]
    }
  }
}
```

Провайдер Kubernetes настраивается с помощью свойств кластера EKS, подготовленного ранее. Эти свойства сообщают Terraform, что для подключения к кластеру необходимо использовать инструмент AWS Authenticator вместе с предоставленным нами сертификатом.

Как упоминалось ранее, мы также используем провайдер для Helm. Диспетчер пакетов Helm — это популярный инструмент описания развертывания

ПО в Kubernetes и распространения приложений Kubernetes в виде пакетов. Он похож на другие инструменты управления пакетами, такие как `apt-get` в мире Linux, и призван упростить и облегчить установку приложений на базе Kubernetes. Чтобы настроить провайдер Helm, достаточно указать несколько параметров подключения к Kubernetes.

Пакеты развертывания Helm называют *чартами* (charts). Для установки сервера Argo CD мы используем пакет Helm, подготовленный сообществом Argo CD. Добавьте код с описанием процесса установки из примера 7.20 в файл `main.tf`.

Пример 7.20. module-argo-cd/main.tf (Helm)

```
resource "kubernetes_namespace" "example" {
  metadata {
    name = "argo"
  }
}

resource "helm_release" "argocd" {
  name       = "msur"
  chart      = "argo-cd"
  repository = "https://argoproj.github.io/argo-helm"
  namespace  = "argo"
}
```

Этот код создает пространство имен для Argo CD и использует провайдер Helm для выполнения установки. Чтобы получить законченный модуль установки Argo CD, осталось лишь определить некоторые переменные.

Переменные для Argo CD

Создайте файл `variables.tf` в репозитории модуля Argo CD и добавьте в него код из примера 7.21.

Пример 7.21. module-argo-cd/variables.tf

```
variable "kubernetes_cluster_id" {
  type = string
}

variable "kubernetes_cluster_cert_data" {
  type = string
}

variable "kubernetes_cluster_endpoint" {
  type = string
}
```

```
variable "kubernetes_cluster_name" {
  type = string
}

variable "eks_nodegroup_id" {
  type = string
}
```

Эти переменные необходимы для настройки провайдеров Kubernetes и Helm в нашем коде. Поэтому их нужно извлечь из выходных данных модуля Kubernetes после его вызова в файле Terraform нашей «песочницы». Прежде чем перейти к этому шагу, отформатируем и проверим написанный код, как мы делали это для других наших модулей:

```
module-argocd$ terraform fmt
[...]
module-argocd$ terraform init
[...]
module-argocd$ terraform validate
Success! The configuration is valid.
```

Убедившись в допустимости кода, зафиксируйте изменения и отправьте их в репозиторий GitHub, чтобы можно было использовать модуль в среде «песочницы»:

```
$ git add .
$ git commit -m "ArgoCD module init"
$ git push origin
```

Теперь, как и раньше, нам просто нужно вызвать этот модуль из определения «песочницы».

Установка Argo CD в «песочнице»

Мы хотим, чтобы установка Argo CD происходила как часть процесса развертывания нашей среды «песочницы», поэтому нужно вызвать модуль из определения Terraform среды «песочницы».

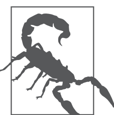
Добавьте код из примера 7.22 в конец файла `main.tf` модуля «песочницы» для установки Argo CD. Не забудьте указать путь к репозиторию вашего модуля на GitHub в свойстве `source`.

Пример 7.22. eenv-sandbox/main.tf (Argo CD)

```
...

# Конфигурация сети
...
```

```
# Конфигурация EKS
...

# Конфигурация GitOps
module "argo-cd-server" {
  source = "*github.com/{YOUR_ARGOCD_MODULE_PATH}*"

  kubernetes_cluster_id        = module.aws-eks.eks_cluster_id
  kubernetes_cluster_name      = module.aws-eks.eks_cluster_name
  kubernetes_cluster_cert_data = module.aws-eks.eks_cluster_certificate_data
  kubernetes_cluster_endpoint  = module.aws-eks.eks_cluster_endpoint
  eks_nodegroup_id = module.aws-eks.eks_cluster_nodegroup_id
}
```

Теперь можно создать тег и отправить файл Terraform в конвейер CI/CD точно так же, как мы делали это раньше. Например, следующая команда поместит тег версии v1.2 в репозиторий и запустит процесс конвейера:



Обязательно дождитесь завершения сборки EKS, прежде чем добавлять тег и фиксировать изменения с описанием установки Argo CD в «песочнице». В противном случае Argo CD просто некуда будет устанавливать из-за отсутствия кластера Kubernetes.

```
$ git add .
$ git commit -m "initial ArgoCD release"
$ git push origin
$ git tag -a v1.2 -m "ArgoCD build"
$ git push origin v1.2
```

Когда конвейер завершит внесение изменений, вы получите сервер GitOps, который поможет развертывать микросервисы. Этим этапом мы завершили определение и подготовку среды «песочницы». Все, что осталось, — протестировать и проверить ее работоспособность.

Тестирование среды

Прежде чем объявить о завершении реализации инфраструктуры, желательно запустить тест и убедиться, что среда подготовлена должным образом. Для этого мы просто попробуем войти в веб-консоль Argo CD. Это докажет, что весь стек запущен и работает. Но, чтобы сделать это, нужно настроить приложение `kubectl` CLI.

Ранее в этой главе, создавая код Terraform для модуля Kubernetes, мы добавили локальный ресурс для создания файла `kubeconfig`. Теперь загрузим этот файл, чтобы иметь возможность подключиться к кластеру EKS с помощью приложения `kubectl`.

Для того чтобы получить файл, перейдите в браузере на вкладку Actions (Действия) в репозитории «песочницы» на GitHub. Вы должны увидеть в верхней части экрана список сборок, созданных в результате запуска конвейера. После выбора только что созданной сборки вы увидите артефакт с именем `kubeconfig`, на котором можно щелкнуть кнопкой мыши, чтобы скачать.

Если у вас возникли проблемы с поиском страницы для загрузки артефакта, то попробуйте следовать инструкциям в документации GitHub (https://oreil.ly/czDRi).

GitHub упакует артефакт в ZIP-файл, поэтому после загрузки вам нужно будет распаковать пакет. Внутри ZIP-файла вы найдете файл `kubeconfig`. Чтобы использовать его, просто создайте переменную среды `KUBECONFIG` и присвойте ей путь к файлу. Это поможет утилите `kubectl` отыскать его. Например, если файл `kubeconfig` находится в каталоге ~/Downloads, то используйте следующее значение:

```
$ export KUBECONFIG=~/Downloads/kubeconfig
```

При желании файл kubeconfig можно скопировать в ~/.kube/config и тогда утилита найдет его без переменной среды. Просто убедитесь, что при этом не затерли конфигурацию Kubernetes, которую уже используете.

Для проверки, что все работает должным образом, выполните следующую команду:

```
$ kubectl get svc
```

В ответ команда должна вывести примерно следующее:

```
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   172.20.0.1   <none>        443/TCP   2h
```

Как видите, сеть и сервис EKS были подготовлены и мы смогли успешно подключиться к кластеру. Чтобы получить эту информацию, `kubectl` вызывает API только что созданного кластера Kubernetes. Получение этого ответа доказывает, что кластер запущен и работает. В качестве заключительного теста проверим, установлен ли в кластере Argo CD.

Выполните следующую команду, чтобы убедиться, что модули pod с Argo CD запущены:

```
$ kubectl get pods -n "argo"
NAME                                                 READY STATUS   RESTARTS
```

```
msur-argocd-application-controller-5bddfb78fc-9jpzj   1/1   Running   0
msur-argocd-dex-server-84cd5fc9b9-bjzrm               1/1   Running   0
msur-argocd-redis-dc867dd9c-rpgww                     1/1   Running   0
msur-argocd-repo-server-75474975cc-j7lws              1/1   Running   0
msur-argocd-server-5cc998b478-wvkrr                   1/1   Running   0
```

Модуль Pod в Kubernetes — это развертываемый модуль, состоящий из одного или нескольких образов контейнеров.

Позже мы используем Argo CD, кластер Kubernetes и остальную инфраструктуру, которую разработали, а теперь, зная, что конвейер и конфигурации работают, уничтожим все это. Но не волнуйтесь: написанный нами код позволит воссоздать среду снова, когда она понадобится.

Очистка инфраструктуры

Теперь у нас есть готовая к работе инфраструктура. Но если вы не планируете использовать ее сразу, то лучше удалить ее, чтобы не нести никаких затрат на ее работу. В частности, аренда общедоступных IP-адресов, которые мы использовали для своей сети, может стоить довольно дорого, если оставить их активными.

Поскольку наша среда теперь полностью определена в декларативных файлах Terraform, мы сможем воссоздать ее, когда она потребуется, поэтому уничтожение существующей среды поможет сократить затраты на нее.

Terraform автоматически уничтожит ресурсы в правильном порядке, поскольку программой внутри был создан граф зависимостей. Чтобы уничтожить среду «песочницы», выполните следующие действия.

1. Перейдите в каталог с кодом среды «песочницы» на вашем компьютере. Это тот же каталог, который вы использовали в пункте «Установка Argo CD в “песочнице”» выше.
2. Извлеките последнюю версию кода из репозитория:

   ```
   env-sandbox$ git pull
   ```

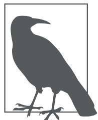
3. Установите провайдеры Terraform, которые используются кодом развертывания среды (они понадобятся для уничтожения ресурсов):

   ```
   env-sandbox$ terraform init
   ```

4. Когда Terraform завершит скачивание плагинов, введите следующую команду, чтобы уничтожить среду «песочницы»:

```
env-sandbox$ terraform destroy
```

5. Terraform покажет ресурсы, которые уничтожит. Выберите ответ `yes`, чтобы продолжить процесс удаления. Весь процесс займет около пяти минут. По завершении все созданные нами ресурсы AWS исчезнут.



Можно уничтожить эти ресурсы AWS с локального компьютера, поскольку у нас есть доступ к AWS и секретные ключи, хранящиеся в локальном файле учетных данных. Но этот способ не подходит для удаления производственной или защищенной среды.

6. Когда процесс удаления завершится, вы увидите примерно такое сообщение:

```
Destroy complete! Resources: 29 destroyed.
```

7. Чтобы убедиться, что ресурсы EKS удалены, выполните следующую команду AWS CLI для получения списка кластеров EKS:

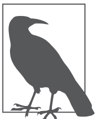
```
$ aws eks list-clusters
```

Вы должны получить ответ, сообщающий, что в вашем экземпляре не осталось кластеров EKS:

```
{
    "clusters": []
}
```

Также можно выполнить следующие команды, чтобы дважды проверить, были ли удалены другие платные ресурсы:

```
$ aws ec2 describe-vpcs --filters Name=cidr,Values=10.10.0.0/16
$ aws elbv2 describe-load-balancers
```



Нет никакой необходимости запускать эти команды CLI, если команда terraform destroy вернула сообщение об успехе. Мы включили их, чтобы вы могли перепроверить, действительно ли они исчезли, чтобы вам неожиданно не выставили счет.

Если что-то пошло не так, используйте консоль AWS и удалите ресурсы вручную. Обратитесь к документации AWS (https://docs.aws.amazon.com), если у вас возникли проблемы с удалением ресурсов через консоль.

Резюме

Мы многое сделали в этой главе. Создали модуль Terraform для построения программно-определяемой сети, охватывающей две зоны доступности в одном регионе. Затем подготовили модуль, создающий экземпляр кластера AWS EKS для Kubernetes. Развернули сервер Argo CD GitOps в кластере, установив пакет Helm. Наконец, реализовали среду «песочницы» в виде кода, который использует все эти модули декларативным, неизменяемым образом.

В данной главе мы подробно остановились на коде Terraform. И сделали это, чтобы вы могли получить представление о том, что требуется для определения среды с использованием инфраструктуры как кода, неизменяемости и конвейера CI/CD. Мы также хотели, чтобы вы ознакомились с шаблоном модуля Terraform и некоторыми проектными решениями, которые вам придется принимать при создании своей инфраструктуры. По мере углубления в разработку развертываемых нами микросервисов могут понадобиться дополнительные модули инфраструктуры. Но позже в книге мы будем просто использовать предварительно написанный код, вместо того чтобы проходить его построчно.

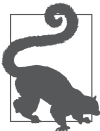
В главе 8 мы вернемся к нашим примерам микросервисов и начнем их разработку. Закончив, мы сможем развернуть их в инфраструктуре, которую только что разработали.

ГЛАВА 8

Рабочее пространство разработчика

В главе 1 мы обсудили, почему архитектура микросервисов обычно наиболее выгодна для сложных систем, и объяснили некоторые из основных причин, подтверждающих это наблюдение.

В любой достаточно сложной системе единственный верный способ направить деятельность участников на позитивное и предсказуемое сотрудничество — сделать правильное поведение абсолютно простым и интуитивно понятным. Если делать «правильно» будет трудно, то со временем большинство людей выберут путь наименьшего сопротивления, который направит их неверной дорогой. Поэтому важно заранее определить воспроизводимые, предсказуемые, стандартизированные процессы разработки, которые позволяют избежать ненужной сложности и создают интуитивно удобную среду для ваших разработчиков.



Инвестирование в исключительный опыт разработчиков с целью получить последовательный и интуитивно понятный подход, позволяющий всем разработчикам легко «поступать правильно», — одно из наиболее недооцененных условий содействия успешной культуре работы с микросервисами.

Вот почему разработка надежных конвейеров непрерывной интеграции и непрерывного развертывания (CI/CD) как для вашего кода, так и для инфраструктуры является ключевым фактором, способствующим успеху в области разработки микросервисов. Вследствие модульного характера архитектуры и акцента на независимом развертывании каждого микросервиса вы получите множество конвейеров. Но не допускайте, чтобы каждая команда создавала конвейер для своего микросервиса по-своему, без какой-либо согласованности с кодовыми

базами других микросервисов. Создание нового микросервиса должно быть быстрым и предсказуемым процессом. В идеале это должен быть шаблонный процесс, большая часть которого автоматизирована.

Надежные конвейеры CI/CD имеют решающее значение, однако не менее важно и то, как настроено локальное рабочее пространство разработки и какие методы используют команды для создания кода. Многие инженеры-программисты пишут большую часть кода на ноутбуках. Включение в этот процесс на ранней стадии и обеспечение правильного руководства и инструментария на данном этапе также могут дать огромные преимущества на более поздних стадиях процесса.

Внесем ясность: мы не имеем в виду, что вы должны диктовать разработчикам каждый аспект рабочего процесса. Например, попытавшись стандартизировать редактор кода, который должны использовать члены любой большой команды, вы быстро наживете много заклятых врагов и ничего не добьетесь. Однако мы можем и должны провозгласить некоторые фундаментальные принципы, которых обязаны придерживаться члены команды.

Эту главу мы начнем с представления набора из десяти весьма категоричных правил и рекомендаций по настройке рабочего пространства разработчика, которые мы с большим успехом использовали в некоторых наших прошлых проектах микросервисов. Далее мы расскажем о настройке локальных контейнерных сред на нескольких платформах и покажем, как запустить стандартный Docker и упрощенный Kubernetes локально. Наконец, мы продемонстрируем продвинутый пример контейнеризации: установку локальной базы данных Cassandra во вновь созданную среду Docker.

К концу этой главы вы получите действующую контейнерную инфраструктуру, готовую к написанию кода некоторых микросервисов. И, что особенно важно, вы получите четкое представление о принципах настройки проектов для простой и интуитивно понятной разработки. Мы используем эти принципы в главе 9 для правильной организации кода, когда перейдем к этапу разработки реализации.

Стандарты программирования и настройки среды разработки

Внедряя какие-либо организационные стандарты, полезно уточнить и согласовать цели, чтобы люди могли понять, «почему» процесс организован именно так, прежде чем им будет представлена фактическая механика, «как» и «что» в нем.

В качестве отправной точки мы рекомендуем следовать трем целям.

- *Код можно настроить за короткий промежуток времени.* Мало что может расстроить больше, чем необходимость тратить много времени на создание среды, позволяющей начать программирование в новой кодовой базе после присоединения к новой команде. Независимо от того, переходите вы в новую компанию или просто спешите протянуть руку помощи другой команде на текущей работе, есть несколько надежных способов избавиться от любого волнения и импульсивности, чем зацикливание на вопросе «Как мне вообще это запустить?». Увы, с такой проблемой приходится сталкиваться слишком часто. Наша цель состоит в том, чтобы новый разработчик, не знакомый с кодом, мог настроить микросервис или набор микросервисов, образующих логическую подсистему, менее чем за час!
- *Новые микросервисы можно создавать быстро, легко и предсказуемо.* Существует много шаблонов, связанных с запуском нового сервиса. Вам нужны соответствующие шаблоны кода для каждого поддерживаемого технологического стека, такого как Java, Go, Node, Python и т. д.; настройка автоматического тестирования и управления данными; зависимости, такие как настроенное хранилище данных; загруженный скелет конвейера. И это лишь малая часть. В наихудшем случае разработчику, запускающему новый микросервис, придется каждый раз выяснять все эти аспекты с нуля.

  На самом деле еще худшим сценарием было бы, если бы разные разработчики без необходимости меняли эти аспекты для схожих микросервисов. Самое эффективное, что можно сделать, чтобы избежать хаоса в большой кодовой базе, — обеспечить согласованность и узнаваемость. Применение хорошо продуманных шаблонов для каждого из стандартных технологических стеков — эффективный способ достижения такой согласованности и высокого качества, а также увеличения скорости разработки.
- *Контроль качества должен быть автоматизирован.* Соблюдение стандартов качества разработки ПО компании следует автоматизировать, оно не должно зависеть от человеческого фактора.

Основываясь на этих целях, можно разработать набор основных рекомендаций по настройке рабочего пространства разработчика.

Десять рекомендаций по организации рабочего пространства разработчика

Нижеследующие рекомендации довольно субъективны и основаны на опыте авторов данной книги. Мы рекомендуем вам использовать эти рекомендации в качестве отправной точки.

Когда у вас накопится опыт создания сервисов в соответствии с этими рекомендациями, то вы сможете подумать об изменении некоторых из них, чтобы они лучше соответствовали вашим индивидуальным потребностям и опыту.

1. *Сделайте Docker единственной зависимостью*. Синдром «у меня работает» преследует многие команды разработчиков. Очень важно, чтобы любой человек мог легко создать такую же среду. Таким образом, сложные ручные настройки должны быть запрещены.

   Мы живем в эпоху контейнеризации, и команды должны использовать это в своих интересах. Для установки кода должна требоваться только среда выполнения Docker и Docker Compose на компьютере — ничего больше! Не должно иметь значения, работает ли компьютер под управлением Windows, macOS или Linux и какие библиотеки присутствуют. Именно такие предположения приводят к неработающим окружениям. Например, не должно быть никаких ожиданий относительно конкретной версии Python, Go, Java и т. д., присутствующей на компьютере разработчика. Инструкции по настройке должны автоматизироваться, а не описываться в файлах `READ.ME`.

2. *Не должно иметь значения, удаленно или локально*. Все должно работать независимо от того, запускает разработчик код на собственном ноутбуке или в облачном сервере с помощью плагинов удаленной разработки в IDE или SFTP. Это условие должно соблюдаться по умолчанию, и если в каком-то случае это невозможно сделать, то причина исключения должна быть обоснована и задокументирована.

3. *Организуйте гетерогенное рабочее пространство*. Хорошее окружение должно поддерживать разработку микросервисов, написанных на нескольких языках программирования, с использованием нескольких систем хранения данных. Архитектура микросервисов предполагает возможность объединения разнородных микросервисов. Это не означает простое размещение одной кодовой базы в одном контейнере или стандартизацию одного технологического стека. Слишком часто мы видим «фреймворк микросервисов [на каком-то языке]» в рекламе. Если 100 % ваших микросервисов написаны на Java, то с окружением явно что-то не так. И да, нет причин для радости, если все ваши сервисы написаны на «крутом» языке, например Go.

   Теперь для протокола: это никоим образом не означает, что в хорошо управляемой среде микросервисов команды должны иметь возможность свободного выбора любого языка и базы данных. Совсем наоборот: когда нет уверенности, проявите осторожность и используйте два, максимум три стека.

Суть в том, что у вас должна быть возможность ввести новый стек, если он действительно нужен. Поэтому в примере настройки и развертывания вы должны показать, что действительно можете это сделать, внедрив несколько стеков.

Правило двух

Мы обнаружили, что активная поддержка гетерогенности в среде микросервисов является отличным подходом. Для каждого критического компонента в системе старайтесь создать по крайней мере две альтернативы, даже если вам достаточно одной. Но обязательно удостоверьтесь, что ваша инфраструктура способна поддерживать два варианта так же легко, как один. Такой подход мы называем «Правило двух» (https://oreil.ly/_vYfU).

Предположим, что большинство ваших API написаны на Node.js — замечательном оптимизированном стеке ввода-вывода для написания API. Посмотрите, могут ли некоторые из них быть реализованы в Go, или Java, или Rust и т. д., возможно, потому, что они делают что-то более привязанное к процессору, в чем Node не силен. Однако пока вы практикуете разнородность, убедитесь, что ограничиваете выбор ваших языков программирования и систем баз данных во всем приложении двумя или тремя. В противном случае рискуете запутать свои команды слишком большим выбором и создать серьезные накладные расходы на сопровождение.

4. *Запуск одного микросервиса и/или подсистемы из нескольких микросервисов должен быть одинаково простым.* Допустим, система бронирования авиабилетов реализована в виде трех микросервисов. Разработчик должен иметь возможность проверить любой из них по отдельности и работать над ним или проверить всю подсистему взаимодействующих микросервисов (реализация системы резервирования) и работать над ней. Обе эти задачи должны быть очень простыми.

5. *Запускайте базы данных локально, если это возможно.* В целях изоляции для любой локальной системы баз данных должны быть предусмотрены альтернативы, действующие в Docker, а переключение на облачные (например, AWS) сервисы должно производиться простым изменением конфигурации. Как пример, в качестве замены S3 можно использовать локальную установку MinIO (https://min.io). Множество альтернативных AWS-сервисов можно установить с сайта GitHub: https://oreil.ly/Lyasd.

6. *Внедряйте руководящие принципы контейнеризации.* Не все подходы к контейнеризации одинаковы. Любой сможет случайно вставить код в контейнер

Docker, но создание контейнерной среды, удобной для разработчиков, требует больших усилий. Ниже перечислены некоторые принципы, которые мы сочли существенными.

А. Несмотря на то что код выполняется в контейнерах, разработчики должны иметь возможность редактировать его на компьютере (например, на своем ноутбуке, сервере разработки EC2) с помощью любого редактора. Однако весь цикл выполнения — запуск/тестирование/отладка — должен проходить в контейнере.

Б. Поскольку Docker Compose может сделать по большей части все то же, что и Dockerfile, разработчики могут легко спутать их. Поэтому важно оговорить, что и когда должно применяться. Мы рекомендуем следующую формулу.

Используйте Dockerfile для создания образа контейнера и Docker Compose для локального запуска, включая сложные интеграции. Образ, созданный с помощью Dockerfile, должен быть доступен для прямого запуска в Kubernetes, AWS ECR, Swarm или любой другой среде выполнения производственного уровня. Обратите внимание, что при этом образ для локальной разработки не обязательно будет таким же, как образ для запуска в производственной среде. Команды часто оптимизируют первый с точки зрения удобства использования, а второй — с точки зрения безопасности и производительности. Хороший пример такого подхода — использование многоступенчатой сборки.

В. Многоступенчатые сборки (https://oreil.ly/qI1Dp) должны применяться в Dockerfiles, чтобы обеспечить использование легковесных образов в производстве и более полнофункциональных образов для локальной разработки.

Г. Пользовательский опыт разработчика имеет решающее значение. Реализация горячей перезагрузки кода и/или возможность подключения отладчика в исходном виде — важная функция.

7. *Установите правила для свободной миграции баз данных*. Чрезвычайно важно управлять базами и содержащимися в них данными таким образом, чтобы поддерживать и улучшать сотрудничество в команде. Изменения в схемах данных должны быть систематизированы и применяться автоматически. Следующий список принципов облегчает свободное управление данными в среде микросервисов.

А. Все и любые изменения в схеме БД должны определяться в виде серии сценариев «миграции базы данных». Файлы миграции должны именоваться и упорядочиваться по дате.

Б. Миграции баз должны поддерживать как изменения схемы, так и вставку образцов данных.

В. Миграция базы данных должна быть частью запуска проекта (с помощью `Make start`, см. следующий раздел) и выполняться принудительно.

Г. Миграция базы данных должна выполняться автоматизированно и быть частью любой сборки (интеграция, сборка функциональных ветвей для PR и т. д.).

Д. Должна иметься возможность указать, какие миграции выполняются в тех или иных средах (или какие из них могут быть пропущены), чтобы, например, миграции, связанные с созданием образцов данных, можно было пропустить в промышленной среде.

Е. Эти правила применимы ко всем системам хранения данных: реляционным, столбчатым, NoSQL и т. д.

Ж. Несколько примеров:

а) Flyway разместили это введение в миграцию баз данных (https://flywaydb.org);

б) посмотрите сообщение в блоге (https://oreil.ly/Vg41z) Дэниела Миранды и др. о миграции баз данных Cassandra;

в) посмотрите пример (https://oreil.ly/EqTxj) применения db-migratesql из Node для базы данных MySQL.

8. *Определите прагматичную практику автоматизированного тестирования.* Автоматизированное тестирование — сложная тема. Мы в своей практике видели обе крайности спектра: одни команды полностью отказываются от автоматизированного тестирования, а другие чрезмерно усердствуют в применении разработки через тестирование до такой степени, что это становится проблемой. Мы же выступаем за взвешенный, прагматичный подход, уравновешивающий опыт разработчиков с показателями качества и учитывающий различные предпочтения отдельных разработчиков в команде.

А. Написание тестов до кода, в процессе и после разработки кода — все эти практики приемлемы, если весь код проверяется с помощью разумного количества тестов до его включения в основную ветвь, прежде чем будет объединен с основной ветвью программы.

Б. Команды должны использовать подходы к тестированию и фреймворки, которые являются идиоматичными для платформ/стеков, используемых в разработке (например, JUnit для Java). Кодовые базы, использующие один и тот же стек (например, Go, Java и т. д.), должны тестироваться

с применением единого подхода. Различные микросервисы на одном и том же языке не должны тестироваться по-разному, независимо от того, кто их написал и когда.

В. Использование внешних инструментов, особенно для приемочных испытаний или оценки производительности, допускается при надлежащем обосновании и соблюдении важного условия: эти инструменты (например, Cucumber) должны быть интегрированы в код/репозиторий самого сервиса, а их использование и запуск должны быть максимально простыми. Не нужно требовать от обычного разработчика сервиса настраивать что-либо, чтобы все заработало, и он должен иметь простую возможность запускать тесты с помощью команды типа `make test-all`.

Г. С особым вниманием и осторожностью следует относиться к автоматизированным тестам, охватывающим границы отдельных микросервисов. Они должны применяться на более высоком уровне (например, на уровне API, который вызывает микросервисы, или пользовательского интерфейса), или в некоторых случаях можно настроить выделенный репозиторий для размещения инструментов оркестрации и автоматизации тестирования.

Д. Необходимо настроить инструменты форматирования/статического анализа кода, а их конфигурации должны быть адаптированы к стилю организации.

9. *Ветвление и слияние*. Практически каждый в наши дни использует ту или иную систему управление версиями кода. Хотя основы разработки, опирающейся на управлении версиями, хорошо понятны, стоит помнить основные принципы чистого ветвления, которые должны соблюдать все члены команды.

   А. Вся разработка должна происходить в функциональных и нестабильных ветвях.

   Б. Слияние кода из нестабильной и основной ветвей не должно допускаться без прохождения всех тестов в этой ветви (включая интеграционные тесты во временном интеграционном кластере, развернутом для ветви).

   В. Результаты тестирования (после каждой фиксации) должны быть легко доступны проверяющим, которые выполняют запросы на извлечение.

   Г. Ошибки форматирования/статического анализа должны препятствовать фиксации кода и/или его объединению с кодом в основной ветви.

10. *В файле makefile должны быть реализованы общие цели*. В каждом репозитории кода (обычно каждый микросервис хранится в отдельном репозитории) должен иметься файл makefile, упрощающий работу с кодом, независимо от

используемого технологического стека и языка программирования. Этот файл должен включать стандартные цели, чтобы независимо от того, какую кодовую базу и на каком языке клонировал разработчик, он мог выполнить команду `make run` для запуска кода и команду `make test` для выполнения автоматических тестов.

Мы рекомендуем определить и реализовать следующие стандартные цели в файлах makefile микросервисов:

- `start` — запуск кода;
- `stop` — остановка выполнения;
- `build` — создание кода (обычно это образ контейнера);
- `clean` — очистка всех кэшей и запуск с нуля;
- `add-module` — добавление модуля;
- `remove-module` — удаление модуля;
- `dependencies` — проверка и установка при необходимости всех модулей, объявленных в разделе управления зависимостями;
- `test` — запуск всех тестов и создание отчета об охвате;
- `tests-unit` — запуск только модульных тестов;
- `tests-at` — запуск только приемочных испытаний;
- `lint` — запуск linter для форматирования в соответствии со стандартными стилями программирования;
- `migrate` — запуск миграции базы данных;
- `add-migration` — создание новой миграции базы данных;
- `logs` — показать журналы (из контейнеров);
- `exec` — выполнить пользовательскую команду внутри контейнера с кодом.

Просмотрите примеры микросервисов в Go (https://oreil.ly/SY_ph) и Node (https://oreil.ly/IMfBj), которые следуют вышеупомянутому шаблону, и пример настройки рабочего пространства для разработки нескольких микросервисов в соответствии с рекомендациями на сайте GitHub (https://oreil.ly/rJyPX). Для удобства мы опубликовали данный шаблон на Github (https://oreil.ly/kd2VT) в формате Markdown, чтобы вы могли ссылаться на него в ваших проектах, если требуется.

Рассмотрим то, что мы уже изучили. Сначала мы признали, что опыт разработчиков имеет первостепенное значение для создания счастливых, высокоэффективных и автономных команд. Затем определили три основные цели, чтобы добиться максимального удобства для разработчиков. И последнее, но

не менее важное: перечислили десять принципов, которые, по нашему опыту, сулят достижение этих целей. В результате мы получили воспроизводимый план создания очень удобных рабочих пространств для разработчиков и команд, независимо от выбора технологического стека или конкретных инструментов. Эта прочная основа поможет вам радовать свои команды и создавать крепкие связи между ними, когда вы начнете строить организацию микросервисов или реорганизовывать существующие команды для перехода к структуре микросервисов.

Один из принципов, который позволяет создавать воспроизводимые, надежные и удобные пространства для разработки, — контейнеризация кода с помощью Docker. В следующем разделе мы погрузимся в настройку надежной контейнерной среды на основных платформах, таких как Linux, macOS и Windows.

Локальная настройка контейнерной среды

Ранее в этой главе мы упоминали, что наличие Docker (и, возможно, утилиты `make` для запуска файлов makefile) должно быть единственным, чего можно ожидать от среды разработчика. Все остальное должно легко устанавливаться из нее. Давайте рассмотрим, как можно получить полный набор инструментов Docker или даже кластер Kubernetes с единственным узлом, если понадобится, на различных платформах.

Установить Docker на компьютер с Linux довольно просто (https://oreil.ly/2jdq6), но какие есть способы заполучить его на компьютере с macOS или Windows?

Когда появились Docker4Mac (https://oreil.ly/gXDWu) и Docker4Windows (https://oreil.ly/oLSzW), они произвели настоящую революцию: привнесли передовые возможности Docker в повседневную рабочую среду, которой пользуется большинство людей. Со временем в них добавилась поддержка Kubernetes, и казалось, что мир не может быть более совершенным для серверного веб-разработчика, особенно для тех, кто переходит на микросервисы.

Самый простой способ установить Docker и Kubernetes в macOS или Windows — использовать все те же Docker4Mac и Docker4Windows. Однако есть и другие варианты, которые могут вам понравиться.

Печальная реальность такова, что повседневное использование Docker4Mac и Docker4Windows может быть довольно сложным. Даже на достаточно современном мощном оборудовании мы сталкивались с высокой загрузкой на процессор и быстрой разрядкой батареи. Для некоторых также может стать

проблемой тот факт, что Docker4Mac позволяет устанавливать только по одному экземпляру Docker и Kubernetes. Если вы много экспериментируете, то, возможно, вам захочется получить больше свободы, чтобы ломать стереотипы.

К счастью, есть альтернативные варианты. Один из них — установка собственных виртуальных машин с помощью VirtualBox или его коммерческих альтернатив. Но опыт показывает, что они тяжелее, чем пакеты Docker4Mac/Win.

Одна из наиболее интересных альтернатив, с которой я недавно начал экспериментировать, — Multipass (https://multipass.run). Это удобный инструмент от компании Canonical, создателей Ubuntu, позволяющий очень быстро запускать хосты Docker на базе Ubuntu на компьютере с macOS или Windows (или даже Linux). Multipass поддерживает ряд базовых виртуальных машин, но, что наиболее важно, по умолчанию использует HyperKit в macOS и Hyper-V в Windows (требуется Windows Pro!), которые, по нашему опыту, более легкие.

Установка Multipass

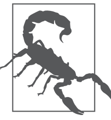
Дистрибутивы Multipass для различных платформ можно скачать с сайта проекта (https://multipass.run). После установки вы сможете поэкспериментировать с некоторыми интересными особенностями, доступными в macOS или Windows Subshell for Linux:

```
→ multipass launch -n docker
Launched: docker
→ multipass list
Name                    State      IPv4           Image
docker                  Running    192.168.64.3   Ubuntu 20.04 LTS
```

По умолчанию Multipass выделяет новой виртуальной машине 1 Гбайт оперативной памяти, 5 Гбайт дискового пространства и одно ядро процессора. Но этого может быть недостаточно. По нашему опыту, при использовании чего-то вроде Node.js или Python с MySQL объема памяти 1 Гбайт может быть достаточно. Но для работы с более тяжеловесными Java-приложениями с системами баз данных на основе Java, такими как Cassandra, вам потребуется больше памяти. Значения по умолчанию можно переопределить при запуске:

```
→ multipass launch -m 4G -n dubuntu
Launched: dubuntu
→ multipass list
Name                    State      IPv4           Image
docker                  Running    192.168.64.3   Ubuntu 20.04 LTS
dubuntu                 Running    192.168.64.4   Ubuntu 20.04 LTS

→ multipass exec dubuntu -- bash
```

```
ubuntu@dubuntu:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3945          79        3640           0         225        3653
```



Multipass позволяет также передать виртуальной машине больше одного процессора, например –c 2; у нас это привело к повреждению контейнеров в macOS. Мы предполагаем, что это может быть как-то связано с ограничениями реализации гипервизора, поэтому будьте осторожны. Увеличение объема памяти не привело к проблемам.

Вы также можете увеличить объем памяти существующего контейнера, не переустанавливая все, что уже настроили. Но будьте осторожны, так как это довольно хрупкий процесс, поэтому лучше остановить процесс Multipass командой `launchctl` (иначе он затрет ваши изменения) и отредактировать конфигурационный JSON-файл, а затем запустить процесс Multipass снова:

```
→ sudo launchctl unload /Library/LaunchDaemons/com.canonical.multipassd.plist
→ sudo vi "/var/root/Library/Application
  Support/multipassd/multipassd-vm-instances.json"
→ sudo launchctl load /Library/LaunchDaemons/com.canonical.multipassd.plist
```

Файл JSON, который вы будете редактировать (`multipassd-vm-instances.json`), должен выглядеть примерно так:

```
{
    "dubuntu": {
        "deleted": false,
        "disk_space": "5368709120",
        "mac_addr": "52:54:00:27:53:b4",
        "mem_size": "4294967296",
        "metadata": {
        },
        "mounts": [
        ],
        "num_cores": 1,
        "ssh_username": "ubuntu",
        "state": 4
    }
}
```

Как нетрудно догадаться, `mem_size` — тот параметр, который вам нужно переопределить (в байтах). Для большей безопасности рекомендуем указывать число, кратное 1 Гбайт. Так как 1 Гбайт — это 1024 × 1024 × 1024 = 1 073 741 824 байта, вам стоит указать число, кратное 1 073 741 824. Например, для 8 Гбайт введите `1073741824 × 8 = 8589934592`.

Вход в контейнер и отображение папок

Вы можете запустить любую команду внутри вашего контейнера с помощью команды типа `multipass exec <имя_контейнера> -- <команда, запущенная внутри>`. Например, вот как можно увидеть объем свободной памяти в контейнере или запустить оболочку Bash:

```
→ multipass exec dubuntu -- free -m
              total        used        free      shared  buff/cache   available
Mem:           3945          77        3640           0         226        3654
Swap:             0           0           0

→ multipass exec dubuntu -- bash

ubuntu@dubuntu:~$ ls -al
total 36
drwxr-xr-x 5 ubuntu ubuntu 4096 .
drwxr-xr-x 3 root   root   4096 ..
-rw------- 1 ubuntu ubuntu  107 .bash_history
-rw-r--r-- 1 ubuntu ubuntu  220 .bash_logout
-rw-r--r-- 1 ubuntu ubuntu 3771 .bashrc
drwx------ 2 ubuntu ubuntu 4096 .cache
drwx------ 3 ubuntu ubuntu 4096 .gnupg
-rw-r--r-- 1 ubuntu ubuntu  807 .profile
drwx------ 2 ubuntu ubuntu 4096 .ssh
ubuntu@dubuntu:~$ exit
exit
→
```

Запуск командной оболочки в основном контейнере можно упростить, предварительно указав, какой из контейнеров должен быть основным. Затем введите команду `multipass shell`:

```
→ multipass set client.primary-name=dubuntu
→ multipass shell

ubuntu@dubuntu:~$
```

Чтобы отобразить домашнюю папку (в macOS) в папку в контейнере, выполните команду:

```
→ multipass mount $HOME dubuntu:/home/ubuntu/mac
Enabling support for mounting -

→ multipass exec dubuntu -- ls -ald mac
drwxr-xr-x 1 ubuntu ubuntu 3936 mac
→ multipass info dubuntu
Name:             dubuntu
State:            Running
```

```
IPv4:           192.168.64.4
Release:        Ubuntu 18.04.4 LTS
Image hash:     2f6bc5e7d9ac (Ubuntu 18.04 LTS)
Load:           0.00 0.08 0.07
Disk usage:     1.1G out of 4.7G
Memory usage:   81.9M out of 3.9G
Mounts:         /Users/irakli => /home/ubuntu/mac
→ multipass exec dubuntu -- ls -al mac
total 240120
drwxr-xr-x 1 ubuntu ubuntu      3936 .
drwxr-xr-x 6 ubuntu ubuntu      4096 ..
-rw-r--r-- 1 ubuntu ubuntu     10244 .DS_Store
drwx------ 1 ubuntu ubuntu        64 .Trash
drwxr-xr-x 1 ubuntu ubuntu       512 .atom
drwxr-xr-x 1 ubuntu ubuntu       128 .aws
```

Теперь, получив с помощью Multipass действующую виртуальную машину с Linux, мы легко можем установить Docker (или даже Kubernetes). Как это сделать, мы покажем в следующем разделе.

Установка Docker

Установка Docker внутри контейнера выполняется как обычно.

```
→ multipass shell

ubuntu@dubuntu:~$ sudo apt-get update && sudo apt-get upgrade -y
ubuntu@dubuntu:~$ sudo apt-get install build-essential -y

# Удаление устаревших версий
ubuntu@dubuntu:~$ sudo apt-get remove docker \
                  docker-ce-cli docker-engine docker.io containerd runc

# Установка Docker и Docker Compose
ubuntu@dubuntu:~$ sudo snap install docker
ubuntu@dubuntu:~$ echo 'export PATH=/snap/bin:$PATH' >> ~/.bashrc
ubuntu@dubuntu:~$ source ~/.bashrc
```

Когда эти шаги будут выполнены, вы получите рабочую версию Docker, но ее можно запустить только от имени root (через `sudo`), а это небезопасно и неудобно. Чтобы исправить проблему, предоставьте непривилегированному пользователю по умолчанию (`ubuntu` для этой версии установки) групповой доступ к Docker, как показано в следующем фрагменте кода. Обратите внимание, что вы должны выйти из Ubuntu и снова войти в систему, чтобы это изменение вступило в силу:

```
ubuntu@dubuntu:~$ sudo groupadd docker
ubuntu@dubuntu:~$ sudo usermod -aG docker $USER
```

```
ubuntu@dubuntu:~$ exit
logout
→ multipass restart
→ multipass shell

ubuntu@dubuntu:~$ docker ps
CONTAINER ID   STATUS   IMAGE   PORTS   NAMES

ubuntu@dubuntu:~$ docker version
Client:
 Version: 19.03.11
 API version: 1.40

ubuntu@dubuntu:~$ $ docker-compose --version
docker-compose version 1.25.5, build unknown
```

Чтобы протестировать новую установку Docker, используем ее для создания базы данных MySQL с Docker Compose.

Тестирование Docker. Для начала создадим файл `mysql-stack.yml` с инструкциями для Docker Compose:

```
version: '3.1'

services:
  db:
    image: mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: rootPass
    ports:
      - 33060:3306
```

Здесь стоит заметить, что по умолчанию этот файл будет называться `docker-compose.yml`, но мы *можем* указать другое имя с помощью специального флага `-f` при запуске Docker Compose. Теперь запустим MySQL в контейнере Docker следующей командой:

```
ubuntu@dubuntu:~$ docker-compose -f mysql-stack.yml up -d

ubuntu@dubuntu:~$ docker ps
CONTAINER ID   STATUS         IMAGE   PORTS
e08f6f072c89   Up 3 seconds   mysql   33060/tcp, 0.0.0.0:33060->3306/tcp
```

Теперь у вас есть рабочие настройки Docker и Docker Compose. В следующем разделе мы покажем, как использовать эти инструменты для простой установки дополнительных компонентов, таких как локальная база данных Cassandra, если вам это потребуется.

Использование локальной версии Docker: установка Cassandra

Мы уже показали, как с помощью Docker Compose запустить базу данных MySQL в контейнере. Теперь рассмотрим более сложный пример: запуск базы данных Cassandra. Учитывая популярность и универсальность Cassandra, вам почти наверняка придется столкнуться с ней на каком-то этапе разработки собственных облачных микросервисов.

Cassandra требует больше 1 Гбайт оперативной памяти, выделяемого по умолчанию, поэтому, запуская контейнер с помощью Multipass, задайте больший объем памяти (например, 6–8 Гбайт).

Сначала создайте файл `docker-compose.yml` со следующим содержимым в любом месте контейнера:

```
version: '3'

services:
  cassandra-seed:
    container_name: cassandra-seed
    image: cassandra:3.11
    ports:
      - "9042:9042"   # Клиенты низкоуровневого протокола
      # - "7199:7199"   # JMX
      # - "9160:9160"   # Клиенты протокола Trift
    volumes:
      - local_cassandra_data_seed:/var/lib/cassandra

volumes:
  local_cassandra_data_seed:
```

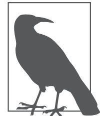
Затем запустите его и проверьте работоспособность:

```
ubuntu@dubuntu:~/cassandra$ docker-compose up -d
Creating network "cassandra_default" with the default driver
Creating cassandra-seed ... done
ubuntu@dubuntu:~/cassandra$ docker-compose ps
Name                   Command               State            Ports
-------------------------------------------------------------------------------
cassandra-seed   docker-entrypoint.sh cassa ...     Up     7000/tcp, 7001/tcp

ubuntu@dubuntu:~/cassandra$ docker exec -it cassandra-seed cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 3.11.6 | CQL spec 3.4.4 | Native protocol v4]
Use HELP for help.
cqlsh> DESCRIBE keyspaces;
```

Последняя команда, `DESCRIBE keyspaces`, покажет все существующие пространства ключей во вновь созданной базе данных Cassandra. Теперь у вас имеется полностью работоспособная локальная версия Cassandra. Далее мы покажем, как установить локальную среду Kubernetes, когда она понадобится.

Установка Kubernetes

В большинстве случаев Docker Compose предоставляет широкие возможности управления компонентами и микросервисами, составляющими ваше приложение. Для более сложных случаев чаще используется фреймворк Kubernetes как предлагающий более широкий спектр возможностей. Однако он имеет пропорционально более высокий уровень сложности.



Избегайте использования Kubernetes локально, пока это не потребуется

Как правило, мы не рекомендуем устанавливать Kubernetes локально для повседневной разработки. Docker и Docker Compose проще использовать для большинства задач, связанных с контейнеризацией, и у них есть более простые инструменты для создания образов контейнеров. Kubernetes отлично подходит для оркестрации парка контейнеров, что редко требуется в среде разработки, но имеет решающее значение для окружений эксплуатации, подготовки к эксплуатации, обкатки, тестирования производительности и т. д. Однако в некоторых обстоятельствах, например для целевого тестирования, может понадобиться использовать Kubernetes локально.

Нельзя просто взять и установить официальный дистрибутив Kubernetes на одну машину. Kubernetes предназначен для развертывания в кластере серверов. Однако существует несколько хороших сторонних проектов, помогающих обойти это требование и установить Kubernetes на одной машине. Наиболее известный из них — Minikube (https://oreil.ly/O9SON), созданный тем же сообществом, которое поддерживает Kubernetes. Однако это не единственное решение. Два других наших фаворита, простых и надежных, — Rancher k3s (https://k3s.io) и Canonical MicroK8s (https://microk8s.io). Чтобы установить Kubernetes локально с помощью k3s, выполните следующую команду:

```
ubuntu@dubuntu:~$ curl -sfL https://get.k3s.io | sh -
[INFO]  Finding release for channel stable
[INFO]  Using v1.17.4+k3s1 as release
[INFO]  Downloading hash \
https://github.com/rancher/k3s/releases/download/v1.17.4+k3s1/...
[INFO]  Downloading binary \
https://github.com/rancher/k3s/releases/download/v1.17.4+k3s1/k3s
[INFO]  Verifying binary download
```

```
[INFO]  Installing k3s to /usr/local/bin/k3s
...

ubuntu@dubuntu:~$ sudo k3s kubectl get nodes
NAME      STATUS   ROLES    AGE    VERSION
dubuntu   Ready    master   104s   v1.17.4+k3s1
```

Установка с помощью MicroK8s выполняется аналогично, но дополнительно нужно добавить текущего пользователя в определенную группу и снова войти в систему, как в случае с Docker:

```
ubuntu@dubuntu:~$ sudo snap install microk8s --classic
microk8s v1.18.1 from Canonical✓ installed
ubuntu@dubuntu:~$ sudo usermod -a -G microk8s $USER
ubuntu@dubuntu:~$ sudo chown -f -R $USER ~/.kube
ubuntu@dubuntu:~$ exit
logout
→ multipass shell

ubuntu@dubuntu:~$ microk8s.kubectl get services --all-namespaces
NAMESPACE  NAME        TYPE       CLUSTER-IP    PORT(S)  AGE
default    kubernetes  ClusterIP  10.152.183.1  443/TCP  3m22s
```

И это по большей части все, что нужно, чтобы получить функционирующую установку Kubernetes на машине разработки.

Как мы упоминали в начале этого раздела, сам фреймворк Kubernetes не содержит инструментов для создания образов контейнеров. Он требует указать URI готового образа в реестре. Из-за данного недостатка Kubernetes оказывается не лучшим вариантом для активной разработки, поскольку не предлагает очевидного решения, позволяющего упростить цикл сборки — запуска — тестирования. Kubernetes на самом деле — это инструмент для сложной оркестрации в средах, не связанных с разработкой (контроль качества, обкатка, подготовка к эксплуатации, эксплуатация и т. д.). Тем не менее через несколько лет после выпуска Kubernetes появился набор инструментов с открытым исходным кодом под названием Skaffold (https://skaffold.dev). Он был разработан с целью включения создания образов контейнеров в жизненный цикл Kubernetes.

Мы не будем использовать локальные версии Kubernetes в большинстве примеров в этой книге. Но если вы хотите взглянуть на пример проекта с открытым исходным кодом, реализующего такой подход, то загляните в репозиторий микросервисов Skaffold (https://oreil.ly/WHcqP), который мы создали в демонстрационных целях.

Резюме

В данной главе мы наметили цели по созданию удобного и эффективного рабочего пространства, в котором разработчики проводят львиную долю своего времени (за исключением, возможно, совещаний). Исходя из этих целей, мы представили десять принципов построения эффективных рабочих пространств и продемонстрировали некоторые шаги по созданию контейнерной базы в основных операционных системах: macOS, Windows и Ubuntu Linux.

Эти концепции и навыки позволят вам создать восхитительную среду для ваших команд разработчиков, сделают адаптацию новых разработчиков приятным занятием и поспособствуют внедрению передовых практик программирования. В главе 9 мы вернемся к изложенным здесь целям и принципам, демонстрируя более мелкие детали процесса объединения кода и базового проекта.

ГЛАВА 9

Разработка микросервисов

Давайте применим некоторые из ранее обсуждавшихся методов и реализуем проект с несколькими микросервисами. Микросервисы в этом проекте будут иметь очень простую реализацию. Мы покажем достаточно кода, чтобы его хватило для демонстрационных целей, но шаги и подходы, которые мы обсудим, можно непосредственно применять к гораздо более крупным реальным проектам.

Сначала мы определим подходящих кандидатов для реализации в виде микросервисов на основе анализа ограниченных контекстов с применением Event Storming — процесса, аналогичного описанному в главе 4. Затем пройдем через семь этапов методологии разработки SEED(S), обсуждавшейся в главе 3, кульминацией которых станет написание кода для двух примеров микросервисов. Разрабатывая эти сервисы, мы будем использовать руководство по моделированию данных, представленное в главе 5. И последнее, но не менее важное: мы покажем, как правильно установить и настроить удобную среду для разработки микросервисов, применив многие рекомендации из главы 8, включая создание зонтичного проекта, позволяющего запустить несколько микросервисов в рабочем пространстве разработчика.

Проектирование конечных точек микросервисов

Предположим, что сеанс Event Storming, проведенный для программного продукта управления бронированием, выявил два основных ограниченных контекста:

- управление информацией о рейсах;
- управление бронированием.

Как обсуждалось в главе 4, на начальных этапах целесообразно проектировать крупномодульные микросервисы. На практике мы часто привязываем их к огра-

ниченным контекстам, то есть нашими первыми двумя микросервисами могут быть ms-flights и ms-reservations!

Теперь, определив целевые микросервисы, применим к ним процесс проектирования SEED(S) (представленный в главе 3). Согласно методологии SEED(S), на первом этапе необходимо определить круг участников. Для наших целей предположим, что основными участниками будут:

- клиент, пытающийся забронировать место на рейсе;
- клиентское приложение авиакомпании (веб-, мобильное и т. д.);
- веб-API, с которыми взаимодействует приложение (в главе 3 мы упоминали, что некоторые называют их BFF API — Backend for Frontend);
- микросервис управления информацией о рейсах: ms-flights;
- микросервис управления бронированиями: ms-reservations.

Рассмотрим примеры заданий для выполнения (Jobs to be done, JTBD), которые наша команда разработчиков могла бы извлечь из опросов клиентов и бизнес-исследований.

1. *Когда* клиент взаимодействует с пользовательским интерфейсом, *приложение должно* отобразить таблицу рассадки с указанием занятых и свободных мест, *чтобы клиент мог* выбрать место.
2. *Когда* клиент завершает бронирование, *веб-приложение должно* зарезервировать для него место, *чтобы приложение могло* избежать случайных конфликтов при бронировании мест.

В главе 3 мы рекомендовали реализовать тонкий слой BFF API без бизнес-логики. Его основная задача — оркестрация микросервисов. Поэтому часто могут иметься задания, для выполнения которых интерфейсу BFF API нужны микросервисы. Следующий список содержит более технические задания и описывает потребности взаимодействия между BFF API и микросервисами.

1. *Когда* API получает запрос на отправку схемы рассадки, ему нужно обратиться к сервису ms-flights и получить схему посадочных мест для рейса, чтобы API мог извлечь информацию о наличии свободных мест и отобразить конечный результат.
2. *Когда* API должен отобразить схему рассадки, *ему нужно* обратиться к сервису ms-reservations и получить список уже забронированных мест, *чтобы API мог* добавить эти данные в схему посадочных мест и вернуть схему рассадки.

3. *Когда* API получает запрос на бронирование места, ему нужно обратиться к сервису ms-reservations, *чтобы API мог* зарезервировать место.

КЛЮЧЕВОЕ РЕШЕНИЕ: ИЗБЕГАТЬ ПРЯМЫХ ВЫЗОВОВ МЕЖДУ МИКРОСЕРВИСАМИ

Обратите внимание, что мы не позволяем сервису ms-flights вызывать сервис ms-reservations для составления схемы рассадки и все взаимодействия выполняем на уровне BFF API. Это одна из рекомендаций, данных в главе 3, согласно которой следует избегать прямых вызовов между микросервисами.

Следуя методологии SEED(S), далее опишем взаимодействия, представленные различными заданиями, используя диаграммы последовательностей UML в формате PlantUML:

```
@startuml

actor Customer as cust
participant "Web App" as app
participant "BFF API" as api
participant "ms-flights" as msf
participant "ms-reservations" as msr

cust -[#blue]-> app ++: "Flight Seats Page"
app -[#blue]-> api ++ : flight.getSeatingSituation()
api -[#blue]-> api: auth
api -> msf ++ : getFlightId()
msf --> api: flight_id
api -> msf: getFlightSeating()
return []flightSeating
api -> msr ++ : getReservedSeats()
return []reservedSeats
return []SeatingSituation
return "Seats Selection Page"
|||

cust -[#blue]->app ++: "Choose a seat & checkout"
app-[#blue]->app: "checkout workflow"
app-[#blue]->api ++: "book the seat"
api -[#blue]->api: auth
api->msr ++: "reserveSeat()"
return "success"
return "success"
return "Success Page"
@enduml
```

Это описание можно отобразить (например, с помощью LiveUML (http://liveuml.com)) в виде диаграммы UML, как показано на рис. 9.1.

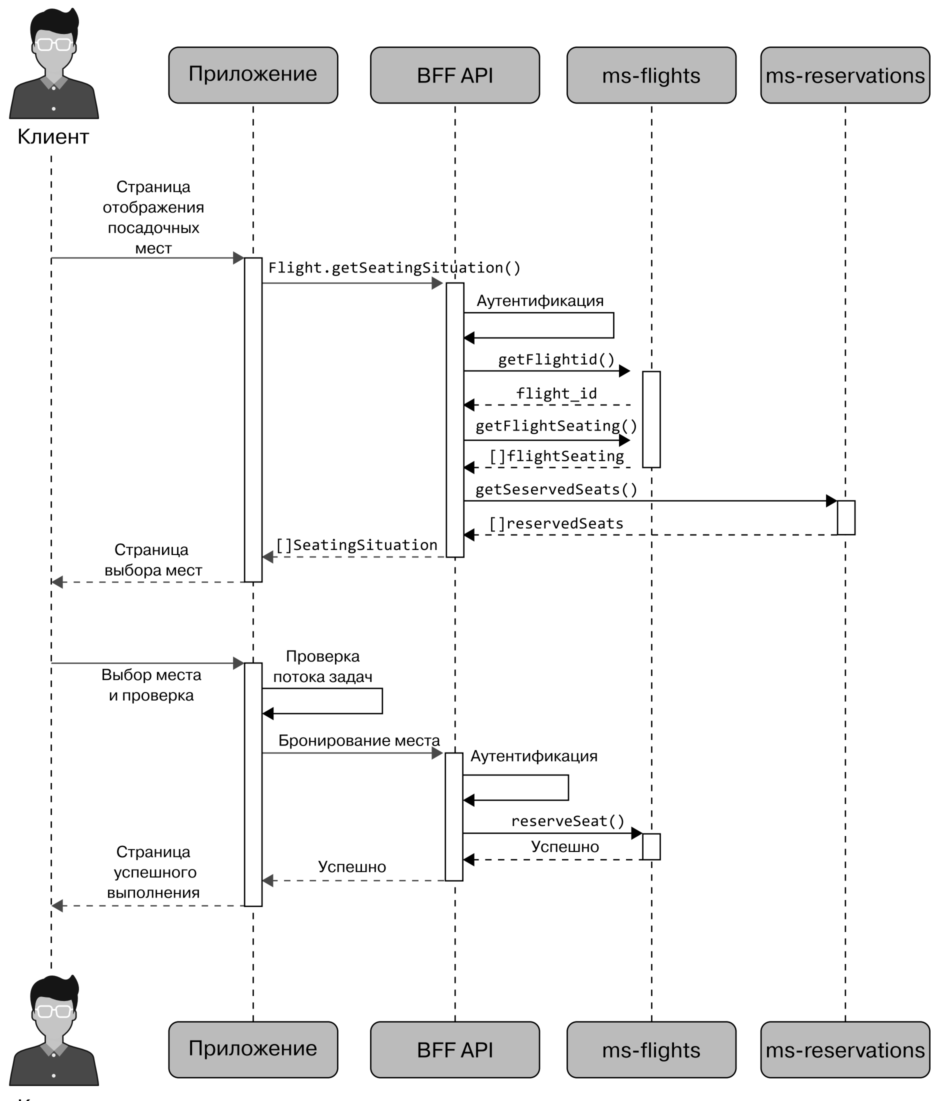


Рис. 9.1. Диаграмма последовательности, представляющая взаимодействия различных JTBD

Как показывает эта диаграмма, первая задача JTBD заключается в том, чтобы предоставить клиенту страницу «места на рейсе». Чтобы выполнить эту задачу, приложению (или сайту) потребуется вызвать BFF API, возвращающий «схему рассадки» — схему посадочных мест с индикаторами, обозначающими, заняты

они или свободны. Сначала API аутентифицирует вызов, чтобы убедиться, что приложение имеет право задавать такие вопросы. Если аутентификация (`auth`) завершилась успехом, то программа попытается получить идентификатор рейса (`flight_id`) из микросервиса ms-flights. Это необходимо, потому что клиенты обычно вводят неуникальный номер рейса (идентифицирующий маршрут, а не конкретный рейс на определенную дату) и дату рейса. Получив уникальный `flight_id`, API попытается получить список мест из сервиса ms-flights. Чтобы показать занятые места, он отдельно запросит у сервиса ms-reservations список забронированных мест.

Особое значение здесь имеет применение принципа, описанного в главе 3, согласно которому микросервисы не должны напрямую вызывать друг друга, а все взаимодействия должны происходить на уровне API. Именно поэтому сервис ms-flights не запрашивает список забронированных мест напрямую у сервиса ms-reservations. Собрав всю необходимую информацию, API сможет вернуть все необходимые данные приложению/сайту, чтобы последний мог отобразить на экране информацию, которая нужна клиенту.

Во второй части рис. 9.1 описана вторая задача JTBD: увидев текущую ситуацию с рассадкой, клиент выбирает конкретное (доступное) место и резервирует его. Чтобы выполнить эту задачу, интерфейсу API снова потребуется выполнить аутентификацию `auth`, а затем вызвать микросервис ms-reservations, возвращающий статус «успешно» или «сбой» в зависимости от результата попытки бронирования. Это позволяет приложению сообщить клиенту, выполнен ли его запрос.

Определив круг задач JTBD и соответствующие им взаимодействия, мы сможем преобразовать их в запросы и действия. Выполним эту процедуру для сервисов ms-flights и ms-reservations. В главе 3 мы объяснили, что также действия и запросы должны разрабатываться не только для микросервисов, но и для BFF API, однако мы оставим эту задачу в качестве упражнения для читателя.

Микросервис управления информацией о рейсах

Определим, какие действия и запросы должен выполнять и обслуживать сервис ms-flights.

Получить информацию о рейсе

- *Входные данные:* `flight_no`, `departure_local_date_time` (формат ISO8601 в локальном часовом поясе).
- *Ответ:* уникальный `flight_id`, идентифицирующий определенный рейс в определенную дату. На практике эта конечная точка, скорее всего, вернет

другие поля, связанные с рейсом, но они не имеют отношения к нашему обсуждению, поэтому мы их опустим.

Получить посадочные места на рейсе (схема мест на рейсе)

- *Входные данные:* `flight_id`.
- *Ответ:* объект Seat Map со схемой мест в формате JSON[1].

Микросервис управления бронированием

Определим, какие действия и запросы должен выполнять и обслуживать сервис ms-reservations.

Запрос зарезервированных мест на рейс

- *Входные данные:* `flight_id`.
- *Ответ:* список номеров занятых мест, номера мест передаются в формате «2A».

Бронирование места на рейсе

- *Входные данные:* `flight_id`, `customer_id`, `seat_num`.
- *Ожидаемый результат:* место резервируется и становится недоступным для других или выдается сообщение об ошибке, если место недоступно для бронирования.
- *Ответ:* успешно (`200 Success`) или сбой (`403 Forbidden`).

Как обсуждалось в главе 3, прелесть определения действий и запросов заключается в том, что они больше приближают нас к созданию технических спецификаций сервисов, чем когда задания представлены в ориентированном на бизнес формате заданий (JTBD).

Теперь, определив действия и запросы, можно приступать к описанию микросервисов, которые мы намерены создать в стандартном формате. В нашем случае создадим микросервисы RESTful и опишем их с помощью OAS. В следующем подразделе мы посмотрим, как может выглядеть эта спецификация для наших двух микросервисов.

---

[1] В демонстрационных целях мы используем структуру объекта Seat Map из Sabre RESTful API (https://oreil.ly/oQA29), представляющего золотой стандарт в этой индустрии.

Проектирование спецификации OpenAPI

Преобразовать только что созданную спецификацию запросов и команд в спецификацию OpenAPI (OpenAPI Specification, OAS) довольно просто. В верхней части спецификации обычно указывается некоторая метаинформация:

```
openapi: 3.0.0
info:
  title: Flights Management Microservice API
  description: |
    API Spec for Flight Management System
  version: 1.0.1
servers:
  - url: http://api.example.com/v1
    description: Production Server
```

В строке запроса к конечной точке `/flights` нужно указать входные параметры `flight_no` и `depar ture_date_time`. Схема также должна описывать структуру ответа JSON, содержащую `flight_id`, код идентификатора аэропорта вылета, код аэропорта назначения и HTTP-код (200) успешного ответа. Эта часть в формате OpenAPI может выглядеть следующим образом:

```
paths:
  /flights:
    get:
      summary: Look Up Flight Details with Flight No and Departure Date
      description: |
        Look up flight details, such as: the unique flight_id used by the
        rest of the Flights management endpoints, flight departure and
        arrival airports.

        Example request:
        ```
          GET http://api.example.com/v1/flights?
              flight_no=AA2532&departure_date_time=2020-05-17T13:20
        ```
      parameters:
        - name: flight_no
          in: query
          required: true
          description: Flight Number.
          schema:
            type : string
          example: AA2532
        - name: departure_date_time
          in: query
          required: true
```

```
        description: Date and time (in ISO8601)
        schema:
          type : string
        example: 2020-05-17T13:20

    responses:
      '200':  # success response
      description: Successful Response
      content:
        application/json:
          schema:
            type: array
            items:
              type: object
              properties:
                flight_id:
                  type: string
                  example: "edcc03a4-7f4e-40d1-898d-bf84a266f1b9"
                origin_code:
                  type: string
                  example: "LAX"
                destination_code:
                  type: string
                  example: "DCA"

            example:
              flight_id: "edcc03a4-7f4e-40d1-898d-bf84a266f1b9"
              origin_code: "LAX"
```

Разрабатывая спецификацию для конечной точки `/flights/{flight_no}/seat_map`, нужно учитывать, что она может принимать входной параметр `flight_no` в самом пути URL, а не в запросе URL. В объекте ответа для демонстрационных целей мы используем структуру объекта Seat Map, которая имитирует структуру золотого стандарта отрасли, Sabre Seat Map API (https://oreil.ly/ySRA0). Если бы вы действительно создавали коммерческий API, то вам пришлось бы разработать свою реализацию или получить разрешение на повторное использование у первоначального автора проекта:

```
/flights/{flight_no}/seat_map:
  get:
    summary: Get a seat map for a flight
    description: |
      Example request:
      ```
        GET http://api.example.com/
            v1/flights/AA2532/datetime/2020-05-17T13:20/seats/12C
      ```
    parameters:
```

```
    - name: flight_no
      in: path
      required: true
      description: Unique Flight Identifier
      schema:
        type : string
      example: "edcc03a4-7f4e-40d1-898d-bf84a266f1b9"

  responses:
    '200':    # success response
      description: Successful Response
      content:
        application/json:
          schema:
            type: object
            properties:
              Cabin:
                type: array
                items:
                  type: object
                  properties:
                    firstRow:
                      type: number
                      example: 8
                    lastRow:
                      type: number
                      example: 23
                    Wing:
                      type: object
                      properties:
                        firstRow:
                          type: number
                          example: 14
                        lastRow:
                          type: number
                          example: 22
                    CabinClass:
                      type: object
                      properties:
                        CabinType:
                          type: string
                          example: Economy
                    Column:
                      type: array
                      items:
                        type: object
                        properties:
                          Column:
                            type: string
```

```
              example: A
            Characteristics:
              type: array
              example:
                - Window
              items:
                type: string
    Row:
      type: array
      items:
        type: object
        properties:
          RowNumber:
            type: number
            example: 8
          Seat:
            type: array
            items:
              type: object
              properties:
                premiumInd:
                  type: boolean
                  example: false
                exitRowInd:
                  type: boolean
                  example: false
                restrictedReclineInd:
                  type: boolean
                  example: false
                noInfantInd:
                  type: boolean
                  example: false
                Number:
                  type: string
                  example: A
                Facilities:
                  type: array
                  items:
                    type: object
                    properties:
                      Detail:
                        type: object
                        properties:
                          content:
                            type: string
                            example: LegSpaceSeat
```

Полный исходный код спецификации можно найти в репозитории GitHub этой книги (https://oreil.ly/Microservices_UpandRunning_api_yml).

Существует несколько редакторов, способных отображать спецификации OAS. Например, на рис. 9.2 показано, как предыдущую спецификацию отображает редактор Swagger Editor (https://editor.swagger.io).

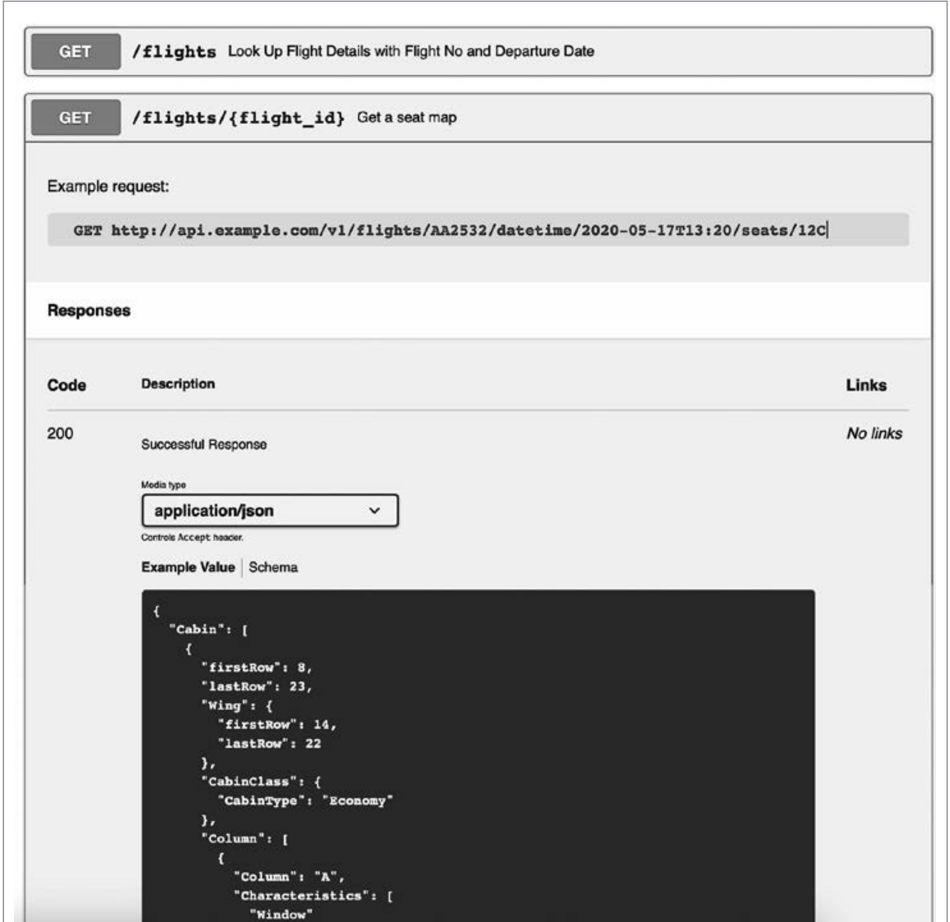


Рис. 9.2. Спецификация OAS для сервиса ms-flights, отображенная с помощью Swagger Editor

Спецификация OAS микросервиса управления информацией о рейсах и конечных точек системы бронирования будет выглядеть аналогично:

```
openapi: 3.0.0
info:
  title: Seat Reservation System API
  description: |
    API Spec for Fight Management System
```

```
  version: 1.0.1
servers:
  - url: http://api.example.com/v1
    description: Production Server
paths:
  /reservations:
    get:
    summary: Get Reservations for a flight
    description: |
      Get all reservations for a specific flight
    parameters:
      - name: flight_id
        in: query
        required: true
        schema:
          type: string
    responses:
      '200':    # success response
        description: Successful Response
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  seat_no:
                    type: string
                    example: "18F"
              example:
                - { seat_no: "18F" }
                - { seat_no: "18D" }
                - { seat_no: "15A" }
                - { seat_no: "15B" }
                - { seat_no: "7A" }
  put:
    summary: Reserve or cancel a seat
    description: |
      Reserves a seat or removes a seat reservation
    requestBody:
      required: true
      content:
        application/json:
            schema:
              type: object
              properties:
                flight_id:
                  description: Flight's Unique Identifier.
                  type : string
                  example: "edcc03a4-7f4e-40d1-898d-bf84a266f1b9"
```

```
                customer_id:
                  description: Registered Customer's Unique Identifier
                  type : string
                  example: "2e850e2f-f81d-44fd-bef8-3bb5e90791ff"
                seat_num:
                  description: seat number
                  type: string
            example:
              flight_id: "edcc03a4-7f4e-40d1-898d-bf84a266f1b9"
              customer_id: "2e850e2f-f81d-44fd-bef8-3bb5e90791ff"
              seat_num: "8D"
    responses:
      '200':
        description: |
          Success.
        content:
          application/json:
            schema:
              type: object
              properties:
                status:
                  type: string
                  enum: ["success", "error"]
                  example:
                    "success"
      '403':
        description: seat(s) unavailable. Booking failed.
        content:
          application/json:
            schema:
              type: object
              properties:
                error:
                  type: string
                description:
                  type: string
              example:
                error: "Could not complete reservation"
                description: "Seat already reserved. Cannot double-book"
```

Разработав проекты сервисов и соответствующие спецификации OAS, можно переходить к последнему шагу в процессе SEED(S): написанию кода микросервисов.

При реализации микросервисов управления информацией о рейсах и бронирования мы применим на практике принципы, рассмотренные ранее в книге. В частности, используем разные технологические стеки, чтобы продемонстрировать возможность поддержки разнородной реализации. Микросервис бронирования мы реализуем на Python и Flask, а микросервис управления информацией о рейсах — на Node/Express.js.

Реализация данных для микросервиса

Чтобы подчеркнуть важность независимости данных, которую мы подробно обсуждали в главе 5, мы не только откажемся от использования общих данных двумя микросервисами, но и намеренно реализуем их с помощью совершенно разных систем управления данными: Redis для бронирования и MySQL для управления информацией о рейсах. Мы также объясним, какие выгоды получает каждый из микросервисов от выбранного механизма хранения данных. Начнем с данных для микросервиса системы бронирования.

Redis для модели данных бронирования

Система бронирования должна иметь возможность фиксировать набор забронированных мест на рейс и резервировать место, только если оно еще не забронировано. В Redis есть идеальная, простая структура данных, которая очень хорошо подходит для нашего варианта использования: *словари* (хеши).

Словари Redis оптимизированы для хранения списков пар «ключ — значение», где ключи и значения имеют строковый тип. Они часто используются для хранения плоских объектов, таких как имя пользователя, фамилия, адрес электронной почты и т. д. Они послужит нам надежным хранилищем информации о бронировании мест. Для каждого `flight_id` (конкретного рейса) можно сохранить в Redis объект словаря, ключами которого являются номера мест на рейсе, а значениями — идентификаторы `customer_id` клиентов, забронировавших их. В Redis есть команды для записи нового значения в словарь и получения всех имеющихся значений (может пригодиться для получения списка забронированных мест), и, что очень удобно, команда, записывающая значение, только если соответствующий ключ (место) отсутствует в словаре. Эта последняя команда идеально нам подходит, поскольку мы не должны допускать двойного бронирования места на рейсе.

> **КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ REDIS ДЛЯ РЕАЛИЗАЦИИ БАЗЫ ДАННЫХ БРОНИРОВАНИЯ**
>
> Использовать Redis как хранилище данных для сервиса бронирования, чтобы задействовать его уникальную простоту и гибкость, характеристики, подходящие для реализации этого микросервиса.

Рассмотрим пример бронирования нескольких мест на рейсе, однозначно идентифицированном с помощью `flight_id` в виде `40d1-898d-bf84a266f1b9`. Если у вас нет установленной версии Redis, воспользуйтесь Redis CLI (https://oreil.ly/ZmFAQ) из рабочего пространства разработки микросервиса бронирования: извлеките код

из репозитория GitHub и вызовите команду `make redis`. После этого вы сможете выполнить следующие команды:

```
> HSETNX flight:40d1-898d-bf84a266f1b9 12B b4cdf96e-a24a-a09a-87fb1c47567c
(integer) 1
> HSETNX flight:40d1-898d-bf84a266f1b9 12C e0392920-a24a-b6e3-8b4ebcbe7d5c
(integer) 1
> HSETNX flight:40d1-898d-bf84a266f1b9 11A f4892d9e-a24a-8ed1-2397df0ddba7
(integer) 1
> HSETNX flight:40d1-898d-bf84a266f1b9 3A 017d40c6-a24b-b6d7-4bb15d04a10b
(integer) 1
> HSETNX flight:40d1-898d-bf84a266f1b9 3B 0c27f7c8-a24b-9556-fb37c840de89
(integer) 1
> HSETNX flight:40d1-898d-bf84a266f1b9 22A 0c27f7c8-a24b-9556-fb37c840de89
(integer) 1
> HSETNX flight:40d1-898d-bf84a266f1b9 22B 24ae6f02-a24b-a149-53d7a72f10c0
(integer) 1
```

Команда `HSETNX`, которую мы используем здесь, присваивает указанное значение заданному ключу, только если он не имеет значения в заданном словаре HSET. Она помогает избежать повторного резервирования мест, которые уже забронированы.

Посмотрим, как можно получить все занятые места на конкретном рейсе:

```
> HKEYS flight:40d1-898d-bf84a266f1b9
1) "12B"
2) "12C"
3) "11A"
4) "3A"
5) "3B"
6) "22A"
7) "22B"
```

Получить ключи вместе с их значениями можно так:

```
> HGETALL flight:40d1-898d-bf84a266f1b9
 1) "12B"
 2) "b4cdf96e-a24a-a09a-87fb1c47567c"
 3) "12C"
 4) "e0392920-a24a-b6e3-8b4ebcbe7d5c"
 5) "11A"
 6) "f4892d9e-a24a-8ed1-2397df0ddba7"
 7) "3A"
 8) "017d40c6-a24b-b6d7-4bb15d04a10b"
 9) "3B"
10) "0c27f7c8-a24b-9556-fb37c840de89"
11) "22A"
12) "0c27f7c8-a24b-9556-fb37c840de89"
13) "22B"
14) "24ae6f02-a24b-a149-53d7a72f10c0"
```

Теперь посмотрим, что получится, если попытаться повторно забронировать уже зарезервированное место, например 12C:

```
> HSETNX flight:40d1-898d-bf84a266f1b9 12C 083a6fc2-a24d-889b-6fc480858a38
(integer) 0
```

Обратите внимание, что эта команда вернула ответ `(integer) 0`, а не `(integer) 1`, как было раньше. Он указывает, что было обновлено 0 полей, потому что место 12C уже зарезервировано.

Как видите, выбор Redis в качестве хранилища данных для сервиса ms-reservations сделал реализацию простой и естественной. Мы получили возможность использовать структуры данных, такие как HSET, в точности отвечающие нашим потребностям. А простая и надежная команда HSETNX позволила нам избежать случайного двойного бронирования.

Redis — фантастическое хранилище ключей и значений, которое можно применять в самых разных ситуациях, именно поэтому у него огромная армия поклонников среди программистов. Однако оно подходит не для всех случаев использования. Иногда у нас могут возникнуть потребности, которые лучше удовлетворяются другими популярными базами данных.

Чтобы продемонстрировать это, в следующем подразделе мы реализуем хранилище данных для микросервиса ms-flights, использовав традиционную базу данных SQL.

Модель данных MySQL для микросервиса управления информацией о рейсах

Первая модель данных, которая нам потребуется, должна содержать схемы посадочных мест. Как мы видели в спецификации OAS для микросервиса управления информацией о рейсах, схема посадочных мест — сложный объект JSON. MySQL лучше подходит для хранения таких данных, чем Redis. В версии MySQL 5.7.8 появилась надежная встроенная поддержка типов данных JSON. Впоследствии она была дополнена и улучшена в версии MySQL 8.x. Теперь MySQL поддерживает также атомарные обновления значений JSON на месте и синтаксис JSON Merge Patch. Для сравнения: Redis поддерживает JSON только с помощью модуля RedisJSON, который не входит в состав стандартного дистрибутива Redis.

Правильно реализованный тип данных JSON дает ощутимые преимущества по сравнению с хранением данных JSON в формате «строка — столбец»: проверка документов JSON во время вставки, внутренне оптимизированное двоичное

хранилище, возможность поиска подобъектов и вложенных значений непосредственно по ключу и т. д.

Кроме того, конечная точка поиска должна позволять запрашивать данные по двум полям: `flight_no` и `datetime`. Реляционная база данных является более естественной структурой для таких запросов. В Redis нам пришлось бы создать составное поле, чтобы добиться того же результата. Технически мы могли бы реализовать этот сервис и с Redis, но есть причины выбрать MySQL и одна из них — возможность продемонстрировать использование различных баз данных для разных сервисов. Очевидно, что в реальной жизни ситуации будут сложнее и потребуют учитывать большее количество аспектов.

Вот как выглядит определение таблицы `seat_maps`:

```
CREATE TABLE `seat_maps` (
  `flight_no` varchar(10) NULL,
  `seat_map` json NULL,
  `origin_code` varchar(10) NULL,
  `destination_code` varchar(10) NULL,

  PRIMARY KEY(`flight_no`)
);
```

Нам также понадобится таблица, отображающая значения `flight_ids` в значения `flight_no` и `datetime`. Вот как может выглядеть ее определение:

```
CREATE TABLE `flights`  (
  `flight_id` varchar(36) NOT NULL,
  `flight_no` varchar(10) NULL,
  `flight_date` datetime(0) NULL,

  PRIMARY KEY (`flight_id`),
  INDEX `idx_flight_date`(`flight_no`, `flight_date`)

  FOREIGN KEY(flight_no)
    REFERENCES seat_maps(flight_no)
);
```

Вставим первую схему посадочных мест:

```
INSERT INTO `seat_maps`(`flight_no`, `seat_map`, `origin_code`, /
`destination_code`) VALUES ('AA2532', '{\"Cabin\": [{\"Row\": [{\"Seat\": /
[{\"Number\": \"A\", \"Facilities\": [{\"Detail\": {\"content\": /
\"LegSpaceSeat\"}}], \"exitRowInd\": false, \"premiumInd\": false, /
\"noInfantInd\": false, \"restrictedReclineInd\": false}], \"RowNumber\": /
8}], \"Wing\": {\"lastRow\": 22, \"firstRow\": 14}, \"Column\": /
[{\"Column\": \"A\", \"Characteristics\": [\"Window\"]}], \"lastRow\": 23, /
\"firstRow\": 8, \"CabinClass\": {\"CabinType\": \"Economy\"}}]}', /
'LAX', 'DCA');
```

Добавив значение JSON в базу данных, мы теперь можем легко выбирать из него определенные вложенные значения или отфильтровать данные. Например:

```
select seat_map->>"$.Cabin[0].firstRow" from seat_maps
```

Теперь, получив рабочую модель данных для обоих микросервисов, мы можем глубже погрузиться в их реализацию.

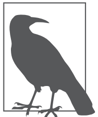
Реализация кода микросервиса

Теперь мы будем работать над достижением второй цели, лежащей в основе «Десяти рекомендаций по организации рабочего пространства разработчика» (см. главу 8), и быстро запустим новые микросервисы, используя проверенные шаблоны для каждого соответствующего технологического стека. Для микросервиса управления информацией о рейсах, реализованного в Node.js, мы используем популярный фреймворк NodeBootstrap (https://nodebootstrap.io). Для микросервиса бронирования на основе Python мы используем репозиторий шаблонов GitHub (https://oreil.ly/g1LIk), содержащий бо́льшую часть необходимого шаблонного кода.

КЛЮЧЕВОЕ РЕШЕНИЕ: НАЧИНАТЬ СОЗДАНИЕ МИКРОСЕРВИСОВ С МНОГОРАЗОВЫХ ШАБЛОНОВ

Применяйте шаблоны кода, чтобы ускорить разработку микросервиса на каждом языке программирования, поддерживаемом в вашей экосистеме. Использование шаблонов помогает ускорить разработку без ущерба для качества и сохраняет единообразие различных микросервисов по их ключевым характеристикам.

Использование любых шаблонов предполагает наличие действующей версии Docker и GNU Make, поскольку мы используем оба компонента, основываясь на десяти рекомендациях. Ничего другого нам не требуется. В главе 8 мы показали, как настроить Docker на нескольких платформах. GNU Make обычно устанавливается по умолчанию в системах macOS и Linux/Unix. Существует несколько способов установить GNU Make в Windows. Один из рекомендованных — подсистема Windows Subsystem for Linux (https://oreil.ly/1JERY).



Код, запускаемый внутри контейнеров, редактируйте на основной машине

Обратите внимание, что во всех дальнейших примерах в этой главе мы предполагаем, что вы работаете на своей основной машине, не выполняя вход в контейнеры Docker.

Когда вы работаете над контейнерным проектом, ваш любимый редактор кода будет установлен на компьютере с macOS, Windows или Linux и вы будете выполнять различные команды `make` на этом компьютере. На этом же компьютере должен быть установлен/доступен Docker, и большинство запускаемых вами команд будет выполняться внутри контейнеров. Но явно запускать командную оболочку в контейнерах потребуется нечасто, если только вы не отлаживаете что-то низкоуровневое.

Код микросервиса управления информацией о рейсах

Чтобы использовать NodeBootstrap для быстрого запуска микросервиса на Node/Express, установите его с помощью `node install -g nodebootstrap` (если у вас уже установлен фреймворк Node) или скопируйте этот репозиторий шаблонов GitHub: https://oreil.ly/Hi-wn.

Первый вариант несколько проще, но мы пойдем вторым путем, поскольку будем считать, что вы не настраивали Node в своей системе. Итак, сделайте шаг вперед и нажмите кнопку Use this template (Использовать этот шаблон) на главной странице репозитория nodebootstrap-microservice, как показано на рис. 9.3.

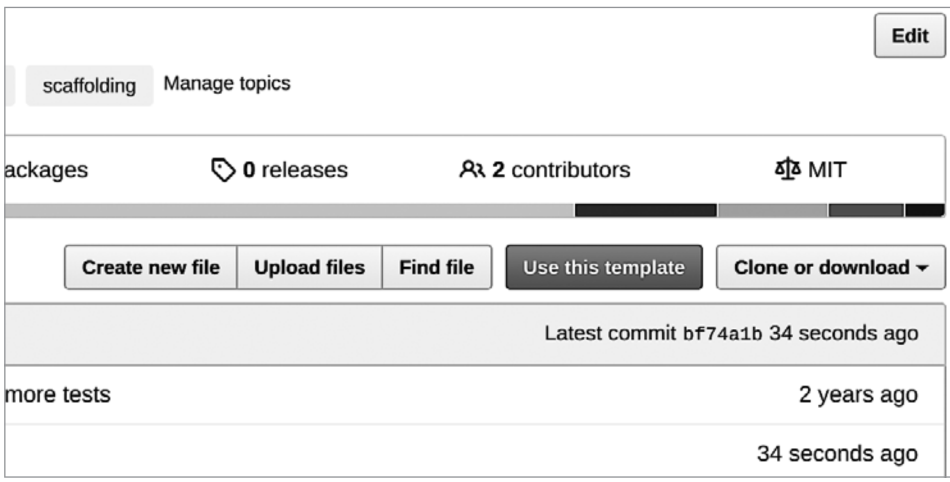


Рис. 9.3. Главная страница репозитория nodebootstrap-microservice

После создания нового репозитория для микросервиса ms-flights в выбранном месте скопируйте его свой компьютер и начнем писать код.

Шаблон NodeBootstrap примечателен тем, что поставляется с полной поддержкой OAS микросервисов. Возьмем спецификацию, которую мы разработали ранее, и поместим ее в файл `docs/api.yml`, заменив образец спецификации, который там уже находится. Для этого перейдите во вложенную папку docs и запустите команду `make start`:

```
→ make start
docker run -d --rm --name ms-nb-docs -p 3939:80 -v \
ms-flights/docs/api.yml:/usr/share/nginx/html/swagger.yaml \
-e SPEC_URL=swagger.yaml redocly/redoc:v2.0.0-rc.8-1
49e0986e318288c8bf6934e3d50ba93537ddf3711453ba6333ced1425576ecdf
server started at: http://0.0.0.0:3939
```

Она преобразует спецификацию в красивый HTML-шаблон и сделает его доступным по адресу http://0.0.0.0:3939. Внешне шаблон, вероятно, будет выглядеть так, как показано на рис. 9.4.

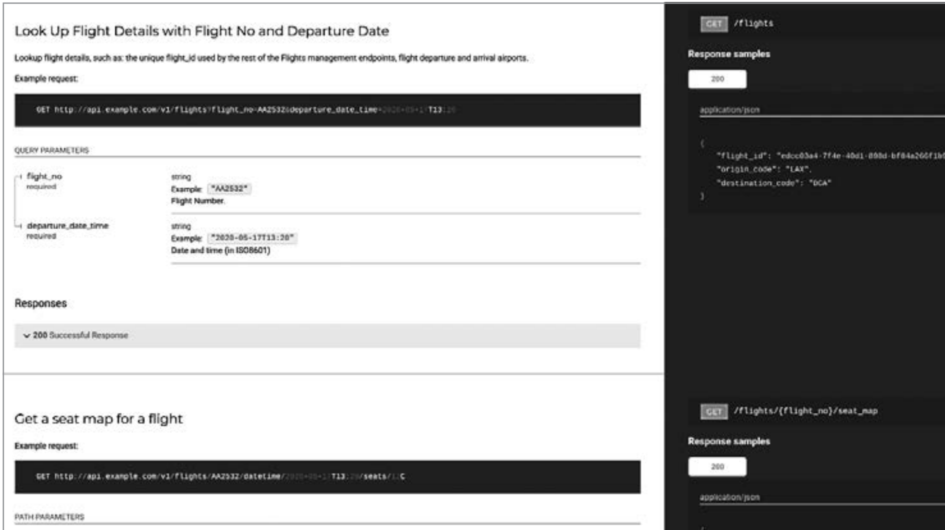


Рис. 9.4. Визуальное представление спецификации OAS микросервиса ms-flights

Микросервис NodeBootstrap включает образец модуля поддержки пользователей в папке lib/users. Поскольку нам нужен не модуль управления пользователями, а модуль управления информацией о рейсах, переименуем эту папку в flights и удалим другой модуль по умолчанию, `lib/homedoc`, так как он нам тоже не понадобится. Удаляя папку lib/homedoc, удалите также ее плагин из `appConfig.js` в корневой папке. Он расположен около строки 24 и выглядит примерно так:

```
app.use('/',     require('homedoc')); // Подключить к корневому пути
```

Аналогично измените подключение для модуля flights в том же файле, чтобы строка выглядела следующим образом:

```
app.use('/flights', require('flights')); // Подключение к подпути
```

Закончив вносить эти изменения, отредактируйте файл `lib/flights/controllers/mappings.js`, чтобы добавить некоторую проверку входных данных и указать, какие функции из модуля actions микросервиса должны вызываться при обращении к каждой из двух конечных точек API:

```
const {spieler, check, matchedData, sanitize} = require('spieler')();

const router      = require('express').Router({ mergeParams: true });
const actions     = require('./actions');

const log = require("metalogger")();

const flightNoValidation = check('flight_no',
  'flight_no must be at least 3 chars long and contain letters and numbers')
  .exists()
  .isLength({ min: 3 })
  .matches(/[a-zA-Z]{1,4}\d+/)

const dateTimeValidation = check('departure_date_time',
  'departure_date_time must be in YYYY-MM-ddThh:mm format')
  .exists()
  .matches(/\d{4}-\d{2}-\d{2}T\d{2}:\d{2}/)

const flightsValidator = spieler([
  flightNoValidation,
  dateTimeValidation
]);
const seatmapsValidator = spieler([
  flightNoValidation
]);

router.get('/', flightsValidator, actions.getFlightInfo);
router.get('/:flight_no/seat_map', seatmapsValidator, actions.getSeatMap);

module.exports = router;
```

Как видите, в этом файле мы настраиваем маршруты для наших двух основных конечных точек и определяем функции, проверяющие наличие и правильность входных параметров. В NodeBootstrap также есть стандартные сообщения об ошибках, помогающие проинформировать клиента об ошибках во входных параметрах.

Теперь реализуем логику. Сначала создадим таблицы MySQL и добавим в них образцы данных. Как вы, наверное, догадались, NodeBootstrap также предостав-

ляет простое решение для этого — в виде миграций: сценариев, реализующих внесение изменений в базы данных, которые позже можно применить в любой среде.

Мы можем создать ряд миграций базы данных с помощью нескольких команд `make` следующим образом:

```
→ make migration-create name=seat-maps
docker-compose -p msupandrunning up -d
ms-flights-db is up-to-date
Starting ms-flights ... done
docker-compose -p msupandrunning exec ms-flights
  ./node_modules/db-migrate/bin/db-migrate create seat-maps --sql-file
[INFO] Created migration at /opt/app/migrations/20200602055112-seat-maps.js
[INFO] Created migration up sql file at
  /opt/app/migrations/sqls/20200602055112-seat-maps-up.sql
[INFO] Created migration down sql file at
  /opt/app/migrations/sqls/20200602055112-seat-maps-down.sql
sudo chown -R $USER ./migrations/sqls/
[sudo] password for irakli:

→ make migration-create name=flights
docker-compose -p msupandrunning up -d
ms-flights-db is up-to-date
ms-flights is up-to-date
docker-compose -p msupandrunning exec ms-flights
  ./node_modules/db-migrate/bin/db-migrate create flights --sql-file
[INFO] Created migration at /opt/app/migrations/20200602055121-flights.js
[INFO] Created migration up sql file
  at /opt/app/migrations/sqls/20200602055121-flights-up.sql
[INFO] Created migration down sql file
  at /opt/app/migrations/sqls/20200602055121-flights-down.sql
sudo chown -R $USER ./migrations/sqls/

→ make migration-create name=sample-data
docker-compose -p msupandrunning up -d
ms-flights-db is up-to-date
ms-flights is up-to-date
docker-compose -p msupandrunning exec ms-flights
  ./node_modules/db-migrate/bin/db-migrate create sample-data --sql-file
[INFO] Created migration at
  /opt/app/migrations/20200602055127-sample-data.js
[INFO] Created migration up sql file at
  /opt/app/migrations/sqls/20200602055127-sample-data-up.sql
[INFO] Created migration down sql file at
  /opt/app/migrations/sqls/20200602055127-sample-data-down.sql
sudo chown -R $USER ./migrations/sqls/
```

После этого нужно открыть соответствующие файлы SQL и вставить содержимое примеров 9.1, 9.2 и 9.3 в каждый из них.

Пример 9.1. /migrations/sqls/[date]-seat-maps-up.sql

```
CREATE TABLE `seat_maps` (
  `flight_no` varchar(10) NOT NULL,
  `seat_map` json NOT NULL,
  `origin_code` varchar(10) NOT NULL,
  `destination_code` varchar(10) NOT NULL,
  PRIMARY KEY (`flight_no`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Пример 9.2. /migrations/sqls/[date]-flights-up.sql

```
CREATE TABLE `flights` (
  `flight_id` varchar(36) NOT NULL,
  `flight_no` varchar(10) NOT NULL,
  `flight_date` datetime(0) NULL,

  PRIMARY KEY (`flight_id`),

  FOREIGN KEY(`flight_no`)
        REFERENCES seat_maps(`flight_no`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

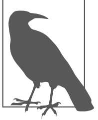
Пример 9.3. /migrations/sqls/[date]-sample-data-up.sql

```
INSERT INTO `seat_maps`
VALUES ('AA2532', '{\"Cabin\": [{\"Row\": [{\"Seat\": [{\"Number\": \"A\",
        \"Facilities\": [{\"Detail\": {\"content\": \"LegSpaceSeat\"}}],
        \"exitRowInd\": false, \"premiumInd\": false, \"noInfantInd\": false,
        \"restrictedReclineInd\": false}], \"RowNumber\": 8}],
        \"Wing\": {\"lastRow\": 22, \"firstRow\": 14},
        \"Column\": [{\"Column\": \"A\", \"Characteristics\": [\"Window\"]}],
        \"lastRow\": 23, \"firstRow\": 8,
        \"CabinClass\": {\"CabinType\": \"Economy\"}}]}', 'LAX', 'DCA');
```

После этого можно просто перезапустить проект командой `make restart`, чтобы автоматически применить миграции (новые миграции применяются при каждом запуске проекта для поддержки согласованности) или явно запустить задачу применения миграций командой `make migrate`.

Из оставшихся модификаций сделайте следующее.

1. Замените в различных файлах название `ms-nodebootstrap-example` на `ms-flights`, если не устанавливали проект с помощью утилиты `nodebootstrap`, и просто клонируйте репозиторий (прежний подход выполнит переименования автоматически).
2. Измените оставшуюся часть исходного кода, чтобы реализовать конечные точки `flights` и `seat_maps` и подключить их к базе данных.



Полный исходный код для сервиса ms-flights

Действующую версию примера микросервиса ms-flights можно найти в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_msflights).

После запуска вы сможете получить доступ к конечной точке `/flights` локально, введя URL, подобный тому, что показан ниже:

```
http://0.0.0.0:5501/flights?flight_no=AA34&departure_date_time=2020-05-17T13:20
```

Конечная точка `seat_maps` должна быть доступна по URL:

```
http://0.0.0.0:5501/flights/AA2532/seat_map
```

Обязательно проверьте все цели в файле makefile. Попробуйте протестировать одну из них, чтобы получить представление о возможностях, которые предлагает шаблонный проект и которые вы должны стремиться предоставить своим разработчикам с помощью своих шаблонов. Чтобы команда `make test` заработала, нужно внести дополнительные изменения, связанные с удалением функциональности из примера проекта.

Мы не будем подробно описывать эти изменения здесь, поэтому просто просмотрите папку /ms-flights в репозитории книги (https://oreil.ly/Microservices_UpandRunning_msflights), где внесены все необходимые изменения. Не стесняйтесь отправлять запросы об ошибках, если столкнетесь с какими-либо проблемами.

Проверки работоспособности

Чтобы управлять жизненным циклом контейнеров, в которых будет развернуто приложение, большинству решений управления контейнерами (например, фреймворку Kubernetes, который мы будем использовать позже в этой книге) требуется сервис, предоставляющий конечную точку для проверки работоспособности. При использовании Kubernetes обычно требуется предоставить конечные точки для проверки работоспособности и готовности к работе.

КЛЮЧЕВОЕ РЕШЕНИЕ: НАЧИНАТЬ СОЗДАНИЕ МИКРОСЕРВИСОВ С МНОГОРАЗОВЫХ ШАБЛОНОВ

Для конечной точки проверки работоспособности мы используем документ RFC (https://oreil.ly/nF9T-) и реализацию на Node.js (https://oreil.ly/ZyfBZ).

Шаблон NodeBootstrap уже содержит пример реализации (https://oreil.ly/EzEIi) этого подхода, нам просто нужно адаптировать его для сервиса ms-flights.

Начнем с замены строк 13–17 в файле `appConfig.js` следующим кодом:

```
// Для проверки работоспособности достаточно значений по умолчанию
const livenessCheck = healthcheck({"path" : "/ping"});
app.use(livenessCheck.express());

// Проверяя готовность, также протестируем БД
const check = healthcheck();
const AdvancedHealthcheckers = require('healthchecks-advanced');
const advCheckers = new AdvancedHealthcheckers();

// Результат проверки работоспособности базы данных кэшируется
// на период 10000 мс = 10 секунд!
check.addCheck('db', 'dbQuery', advCheckers.dbCheck,
  {minCacheMs: 10000});
app.use(check.express());
```

Этот код реализует проверку «Жив ли я?» в конечной точке `/ping` (известную как *проверка работоспособности* или *liveness probe* в Kubernetes) и более продвинутую проверку «Готова ли база данных? Могу ли я на самом деле выполнять полезные операции?» (известную как *проверка готовности* или *readiness probe* в Kubernetes) в конечной точке `/health`. Использовать две проверки для оценки общего состояния очень удобно, поскольку способность микросервиса откликаться не всегда означает, что он полностью работоспособен. Если он зависит от базы данных, которая еще не запустилась или отключена, то на самом деле микросервис не будет готов к полезной работе.

Четвертый аргумент `{minCacheMs: 10000}` в вызове метода `.addCheck()` устанавливает минимальную продолжительность хранения информации в кэше на стороне сервера в миллисекундах. Согласно этому значению промежуточное ПО проверки работоспособности (используемый нами модуль) будет запускать дорогостоящую проверку работоспособности базы данных MySQL не чаще, чем каждые 10 секунд (10 000 миллисекунд)!

Даже если инфраструктура проверки работоспособности (например, Kubernetes) будет вызывать конечную точку проверки работоспособности чаще, промежуточное программное обеспечение будет выполнять только те проверки, которые вы считаете достаточно легковесными. В ответ на запросы для более тяжеловесных проверок (например, базы данных MySQL) промежуточное ПО (модуль Maikai) будет возвращать кэшированные значения, чтобы исключить избыточную нагрузку на нижестоящие системы, такие как база данных. Для того чтобы

завершить настройку, также отредактируйте файл `libs/healthchecksadvanced/index.js` и переименуйте функцию в `dbCheck`. Затем обновите SQL-запрос, чтобы строки 7–10 выглядели так:

```
async dbCheck() {
  const start = new Date();
  const conn = await db.conn();
  const query = 'select count(1) from seat_maps';
```

Если все было сделано правильно, то после запуска микросервиса команда `curl http://0.0.0.0:5501/health` вернет результат, полученный от конечной точки проверки работоспособности, который выглядит следующим образом:

```
{
  "details": {
    "db:dbQuery": {
      "status": "pass",
      "metricValue": 15,
      "metricUnit": "ms",
      "time": "2020-06-28T22:32:46.167Z"
    }
  },
  "status": "pass"
}
```

Если вместо этого выполнить команду `curl http://0.0.0.0:5501/ping`, то вы получите более простой результат:

```
{ "status": "pass" }
```

Если вы столкнетесь с какими-либо проблемами после самостоятельного изменения кода, то ознакомьтесь с полной реализацией микросервиса в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_msflights).

Теперь, получив функционирующий микросервис ms-flights, реализованный с помощью Node.js и MySQL, мы можем перейти к коду, лежащему в основе микросервиса ms-reservations.

Ввод второго микросервиса в проект

Второй микросервис (ms-reservations) мы реализуем на Python и Flask, используя хранилище данных Redis. И снова, следуя второй цели из «Десяти рекомендаций по организации рабочего пространства разработчика» (см. главу 8), мы воспользуемся репозиторием шаблонов GitHub (https://oreil.ly/rjRhK) для стека Python/Flask.

Как нетрудно заметить, этот шаблон во многом похож на шаблон NodeBootstrap, который мы только что использовали для сервиса ms-flights: он имеет всего две зависимости, Docker и make, и содержит в файле makefile все цели для поддержки разработки. Он также включает рабочие настройки для общих задач, таких как тестирование, форматирование и т. д. Однако этому шаблону не хватает одной вещи — поддержки миграции баз данных.

В отличие от MySQL, Redis не использует схемы баз данных, поэтому нет острой необходимости определять различные структуры данных для создания «таблиц». Вы все так же можете использовать миграции для создания тестовых данных в различных средах, но мы оставим эту задачу вам, чтобы вы могли разобраться и получить удовольствие от ее решения. Это одна из особенностей, отличающих данный шаблон от других, в которых используются базы данных SQL.

Как и в случае с сервисом ms-flights, начнем изменение кода с размещения соответствующих спецификаций OAS, которые разработали ранее в этой главе, в каталоге `docs/api.yml` нового репозитория ms-reservations. После запуска `make start` в папке docs (примечание: это отдельный файл makefile, отличный от основного!) вы должны увидеть спецификацию API для сервиса бронирования, отображаемую на http://0.0.0.0:3939 (рис. 9.5).

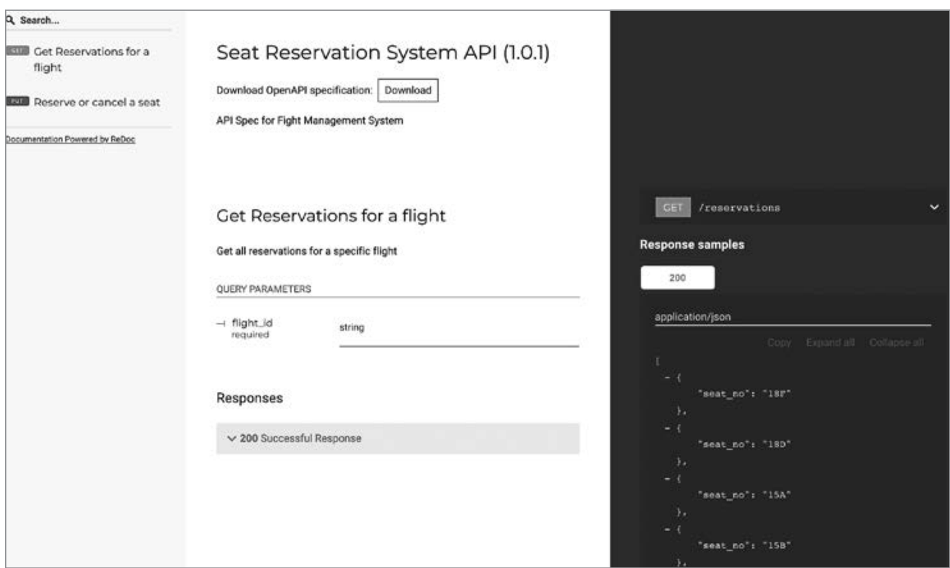


Рис. 9.5. Спецификация OAS микросервиса ms-reservations

Начнем изменять шаблон микросервиса с реализации конечной точки бронирования.

Откройте `service.py` и замените подключение для конечной точки `update_user` `POST /users` на `PUT /reservations`, как показано ниже:

```
@app.route('/reservations', methods=['PUT'])
def reserve():
    """Конечная точка, бронирующая посадочное место для покупателя"""
    json_body = request.get_json(force=True)
    resp = handlers.reserve(json_body)
    if (resp.get("status") == "success"):
        return jsonify(resp)
    else:
        return Response(
            json.dumps(resp),
            status=403,
            mimetype='application/json'
        )
```

Как видите, в зависимости от результата бронирования конечная точка возвращает сообщение об успехе или ошибке и соответствующий код ошибки HTTP.

Чтобы полностью реализовать эту конечную точку, необходимо создать обработчик (обычно ему поручается проверка ошибок, но для простоты мы опустим ее) и добавить ссылку на него в `rc/handlers.py`.

Мы сделаем это, заменив обработчик создания пользователя `save_user` следующим:

```
def reserve(json_body):
    """Сохранение обратного вызова бронирования"""
    return model.save_reservation(json_body)
```

Самое главное: нам нужно сохранить информацию о бронировании в базе данных, для чего заменим функцию `save_user` в `src/models.py` следующим кодом:

```
def save_reservation(reservation):
    """Сохранение бронирования в базе данных Redis"""

    seat_num = reservation['seat_num']
    try:
        result = this.redis_conn.hsetnx(
            this.tblprefix + reservation['flight_id'],
            seat_num,
```

```
            reservation['customer_id'])
    except redis.RedisError:
        response = {
            "error" : f"Unexpected error reserving {seat_num}"
        }
        log.error(f"Unexpected error reserving {seat_num}", exc_info=True)
    else:
        if result == 1:
            response = {
                "status": "success",
            }
        else:
            response = {
                "error" : f"Could not complete reservation for {seat_num}",
                "description" : "Seat already reserved. Cannot double-book"
            }

    return response
```

Этот код реализует ту же команду `hsetnx` на Python, которую мы вручную выполняли ранее в Redis CLI, когда демонстрировали преимущества использования Redis для модели данных микросервиса бронирования. Метод Redis `hsetnx` присваивает заданному ключу значение, только если он еще не определен. Именно так мы избегаем случайного двойного бронирования. Когда `hsetnx` отклоняет попытку, обнаружив точно такой же ключ в базе данных, он возвращает `0` (сообщая, что «изменено 0 записей»); в противном случае возвращает `1`. Так он сообщает о наличии или отсутствии конфликта.

Вы также должны объявить префикс уровня таблицы для резервирования в области модуля, добавив в `src/models.py` следующий код в районе строки 19, сразу после объявления `this = sys.modules[__name__]`:

```
this = sys.modules[__name__]  # Существующая строка
this.tblprefix = "flights:" # Новая строка
```

Шаблон микросервиса, который мы использовали, уже содержит весь код, необходимый для получения соответствующих учетных данных и конфигурации из среды и подключения к базе данных Redis. Он реализован в соответствии с правилами, изложенными в манифесте по созданию улучшенных облачных приложений, известном как «Приложение двенадцати факторов» (https://12factor.net/config). В частности, шаблон соответствует третьему правилу, в котором определяются предпочтительные способы управления конфигурацией. Сам факт, что в примененном нами шаблоне эта рекомендация уже полностью реализована, еще раз демонстрирует преимущества использования шаблонов для разработки микросервисов.

После внесения всех необходимых изменений конечная точка должна заработать. Ее можно запустить командой `make` на верхнем уровне исходного кода, которая создаст и запустит проект на 0.0.0.0:7701.

Если вы столкнетесь с проблемами или по какой-либо причине захотите просмотреть журналы приложения, то используйте цель `make logs-app`:

```
→ make logs-app
docker-compose -p ms-workspace-demo logs -f ms-template-microservice
Attaching to ms-template-microservice
ms-template-microservice    | [INFO] Starting gunicorn 20.0.4
ms-template-microservice    | [INFO] Listening at: http://0.0.0.0:5000 (1)
ms-template-microservice    | [INFO] Using worker: sync
ms-template-microservice    | [INFO] Booting worker with pid: 15
```

Как вы могли заметить, в журнале указано, что сервис запущен на порте 5000, но это порт внутри контейнера Docker, а не на самом компьютере! Мы отобразили стандартный порт Flask 5000 в порт 7701 на компьютере. Объединенные журналы приложения и базы данных можно просмотреть, выполнив команду `make logs`, а журналы базы данных — выполнив команду `make logs-db`.

Теперь выполним несколько команд `curl`, чтобы забронировать пару мест:

```
curl --header "Content-Type: application/json" \
  --request PUT \
  --data '{"seat_num":"12B","flight_id":"werty", "customer_id": "dfgh"}' \
  http://0.0.0.0:7701/reservations

curl --header "Content-Type: application/json" \
  --request PUT \
  --data '{"seat_num":"12C","flight_id":"werty", "customer_id": "jkfl"}' \
  http://0.0.0.0:7701/reservations
```

Мы также можем протестировать нашу защиту от случайного двойного бронирования. Проверим ее, попытавшись забронировать уже забронированное место (например, 12C):

```
curl -v --header "Content-Type: application/json" \
  --request PUT \
  --data '{"seat_num":"12C","flight_id":"werty", "customer_id": "another"}' \
  http://0.0.0.0:7701/reservations
```

В ответ мы получили `HTTP 403` и сообщение об ошибке:

```
→ curl -v --header "Content-Type: application/json" \
>   --request PUT \
>   --data '{"seat_num":"12C","flight_id":"werty", "customer_id": "another"}' \
>   http://0.0.0.0:7701/reservations
```

```
*   Trying 0.0.0.0:7701...
* TCP_NODELAY set
* Connected to 0.0.0.0 (127.0.0.1) port 7701 (#0)
> PUT /reservations HTTP/1.1
> Host: 0.0.0.0:7701
> User-Agent: curl/7.68.0
> Accept: */*
> Content-Type: application/json
> Content-Length: 64
>
< HTTP/1.1 403 FORBIDDEN
< Server: gunicorn/20.0.4
< Connection: close
< Content-Type: application/json
< Content-Length: 111
<
* Closing connection 0
{"error": "Could not complete reservation for 12C",
"description": "Seat already reserved. Cannot double-book"}
```

Идеально!

Теперь, имея некоторые данные в хранилище Redis, можно приступить к реализации конечной точки поиска забронированных мест. И снова начнем с определения отображения в `service.py`, заменив конечную точку приветствия по умолчанию `/hello/<name>` на следующую:

```
@app.route('/reservations', methods=['GET'])
def reservations():
    """Получение конечной точки бронирования"""
    flight_id = request.args.get('flight_id')
    resp = handlers.get_reservations(flight_id)
    return jsonify(resp)
```

Обработчик в `src/handlers.py` снова будет иметь простую реализацию, так как мы опускаем проверку входных данных ради краткости:

```
def get_reservations(flight_id):
    """Получение обратных вызовов бронирования"""
    return model.get_reservations(flight_id)
```

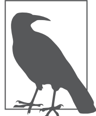
Код модели будет выглядеть следующим образом:

```
def get_reservations (flight_id):
    """Список забронированных мест на рейс, из БД Redis"""
    try:
        key = this.tblprefix + flight_id
        reservations = this.redis_conn.hgetall(key)
    except redis.RedisError:
        response = {
            "error" : "Cannot retrieve reservations"
        }
```

```
        log.error("Error retrieving reservations from Redis",
            exc_info=True)
    else:
        response = reservations

    return response
```

Чтобы протестировать эту конечную точку, выполним команду `curl` и проверим, что получили ожидаемый ответ JSON:

```
→ curl -v  http://0.0.0.0:7701/reservations?flight_id=werty
*    Trying 0.0.0.0:7701...
* TCP_NODELAY set
> GET /reservations?flight_id=werty HTTP/1.1
> Host: 0.0.0.0:7701
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: gunicorn/20.0.4
< Connection: close
< Content-Type: application/json
< Content-Length: 90
<
{
  "12B": "dfgh",
  "12C": "jkfl",
}
* Closing connection 0
```



Полный исходный код для сервиса ms-reservations

Действующую версию примера микросервиса ms-reservations можно найти в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_msreservations).

Просмотрите код и попробуйте использовать различные цели `make`, доступные в репозитории, чтобы лучше понять, что дает шаблон, из которого был получен этот код.

Также используйте эту возможность, чтобы сделать перерыв и похвалить себя — вы только что создали и запустили два микросервиса идеального размера, безупречно реализованных с применением двух разных технологичнских стеков! Ура!

Теперь нам нужно найти способ запустить эти два микросервиса (и любые дополнительные компоненты, которые вы можете создать в будущем) как единое целое. Для этого мы введем понятие зонтичного проекта и объясним, как его разработать.

Подключение сервисов к зонтичному проекту

Разработка отдельных микросервисов — то, на что команды должны тратить бо́льшую часть своего времени. Это важно для автономии команды, которая влечет минимизацию координации, а, как мы уже говорили, основная наша работа по проектированию системы в стиле микросервисов должна быть направлена на минимизацию потребностей в координации. Тем не менее в какой-то момент нам потребуется запустить весь проект — все микросервисы, работающие вместе. Даже притом, что такая необходимость возникает относительно редко, очень важно упростить ее, поэтому в четвертом принципе «Десяти рекомендаций по организации рабочего пространства разработчика» (см. главу 8) говорится: «Запуск одного микросервиса и/или подсистемы из нескольких микросервисов должен быть одинаково простым».

Нам нужен простой в использовании *зонтичный проект*, позволяющий запустить все микросервисы одной простой командой и заставляющий их слаженно работать вместе до тех пор, пока мы не решим остановить проект со всеми его компонентами. Очевидно, что это действие тоже должно быть реализовано очень простым. Чтобы наши разработчики не допускали ошибок, все действия должны быть простыми!

Чтобы развернуть простой в применении зонтичный проект, мы используем шаблон рабочего пространства разработки микросервисов, доступный на сайте GitHub (https://oreil.ly/VpyDJ), и создадим рабочее пространство для себя, реализацию которого можно найти в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_workspace).

КЛЮЧЕВОЕ РЕШЕНИЕ: ПРИМЕНЯТЬ FAUX GIT SUBMODULES

Для получения кода отдельных микросервисов в рамках зонтичного репозитория мы используем проект с открытым исходным кодом Faux Git Submodules (https://oreil.ly/ic_c0). Идея состоит в том, чтобы упростить переход в подпапку репозитория рабочего пространства, содержащую микросервис, и рассматривать его как полноценный репозиторий, позволяющий изменять, фиксировать и выгружать код. Основная цель идентична цели обычных подмодулей Git за исключением того, что любой, кто их использовал, знает, что обычные подмодули могут вести себя непредсказуемо и часто доставляют немало проблем. Подмодули Faux, на наш взгляд, намного проще и работают более предсказуемо.

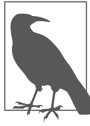
Начнем с определения ссылок на два только что созданных репозитория в новом рабочем пространстве, отредактировав файл `fgs.json`, чтобы он выглядел примерно так:

```
{
  "ms-flights" : {
    "url"  : "https://github.com/implementing-microservices/ms-flights"
  },

  "ms-reservations" : {
    "url" : "https://github.com/implementing-microservices/ms-reservations"
  }
}
```

Здесь мы определили ссылки на репозитории ms-flights и ms-reservations, использовав протокол http://, доступный только для чтения. Это сделано для того, чтобы вы могли следовать примеру. В реальных проектах лучше определять ссылки с использованием протокола git://, допускающего возможность чтения/записи, чтобы иметь возможность их изменять.

Теперь, настроив `repos.json`, перенесем микросервисы ms-flights и ms-reservations в рабочее пространство:

```
→ make update
git clone -b master \
  https://github.com/implementing-microservices/ms-flights ms-flights
Cloning into 'ms-flights'...

git clone -b master \
  https://github.com/implementing-microservices/ms-reservations ms-reservations
Cloning into 'ms-reservations'...
```



Эта операция также помогает добавить извлеченные репозитории в файл .gitignore родительской папки, чтобы родительский репозиторий не пытался дважды выполнять фиксации в неправильное место.

Нам также нужно отредактировать сценарии `bin/start.sh` и `bin/stop.sh` для внесения изменений по умолчанию. Отредактируйте `bin/start.sh`, как показано в примере 9.4.

Пример 9.4. bin/start.sh

```
#!/usr/bin/env bash
set -eu

export COMPOSE_PROJECT_NAME=msupandrunning

pushd ms-flights && make start
popd
pushd ms-reservations && make start
popd

make proxystarts
```

Отредактируйте `.bin/stop.sh`, как показано в примере 9.5.

Пример 9.5. bin/stop.sh

```
#!/usr/bin/env bash
set -eu

export COMPOSE_PROJECT_NAME=msupandrunning

pushd ms-flights && make stop
popd

pushd ms-reservations && make stop
popd

make proxystop
```

Для простоты, но без потери автоматизации, в нашем рабочем пространстве используется пограничный маршрутизатор Traefik (https://oreil.ly/I-ddh), обеспечивающий беспрепятственную маршрутизацию к микросервисам. Он устанавливается файлом `docker-compose.yml` (https://oreil.ly/vBvwS). Кроме того, нам нужно добавить метки для Traefik в файлы `docker-compose.yml` обоих микросервисов, чтобы обеспечить правильную маршрутизацию этих сервисов, как показано в примерах 9.6 и 9.7.

Пример 9.6. ms-flights/docker-compose.yaml

```
services:
  ms-flights:
    container_name: ms-flights
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.ms-flights.rule=PathPrefix(`/reservations`)"
```

Пример 9.7. ms-reservations/docker-compose.yaml

```
services:
  ms-reservations:
    container_name: ms-reservations
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.ms-reservations.rule=PathPrefix(`/reservations`)"
```

Нам также необходимо изменить имя зонтичного проекта (служащее пространством имен и сетевым именем для всех сервисов) в файле makefile рабочего пространства. Для этого измените строку `project:=ms-workspace-demo`, как показано ниже:

```
project:=msupandrunning
```

После запуска рабочего пространства командой `make start` на уровне рабочего пространства вы получите доступ к обоим микросервисам в форме, прикрепленной к рабочему пространству. Мы подключили Traefik к локальному порту 9080, сделав http://0.0.0.0:9080/ своим базовым URI. Соответственно, запросить системы бронирования и рейсов можно будет так:

```
> curl http://0.0.0.0:9080/reservations?flight_id=qwerty
> curl \
http://0.0.0.0:9080/flights?\
flight_no=AA34&departure_date_time=2020-05-17T13:20
```

Полный исходный код зонтичного проекта доступен в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_workspace).

Резюме

В этой главе мы собрали воедино множество рекомендаций по проектированию системы и реализации кода, разработанные нами, чтобы обеспечить сквозную реализацию пары микросервисов вместе с общим рабочим пространством. Такое решение позволяет работать над сервисами как по отдельности, так и вместе. Мы сделали это путем пошаговой реализации методологии SEED(S) и разработки отдельных моделей данных. Кроме того, мы узнали, как быстро запускать реализации кода из надежных шаблонных проектов.

Способность быстро и эффективно собирать модульные компоненты может существенно повлиять на вашу способность успешно запускать проекты микросервисов. Есть большая разница между вами, сумевшими достичь многого в этой главе, и теми, кто тратит недели на изучение базового шаблона или погружается в кроличью нору неправильных решений. Эта разница может обусловить успех или провал всей инициативы разработки.

ГЛАВА 10

Выпуск микросервисов

Мы подходим к самой захватывающей части процесса создания микросервисов — к моменту, когда все объединяется в одну структуру. К данному моменту мы создали операционную модель, проект микросервиса, основу инфраструктуры и два работающих микросервиса. Теперь возьмем все эти фрагменты и соберем их воедино.

В этой главе мы рассмотрим множество вопросов. Создадим новую среду инфраструктуры под названием *«обкатка»* (staging). Затем добавим в репозиторий кода процесс доставки контейнеров. А когда контейнеры будут готовы к работе, реализуем процесс развертывания с помощью инструмента Argo CD GitOps. По окончании мы получим архитектуру, похожую на изображенную на рис. 10.1.

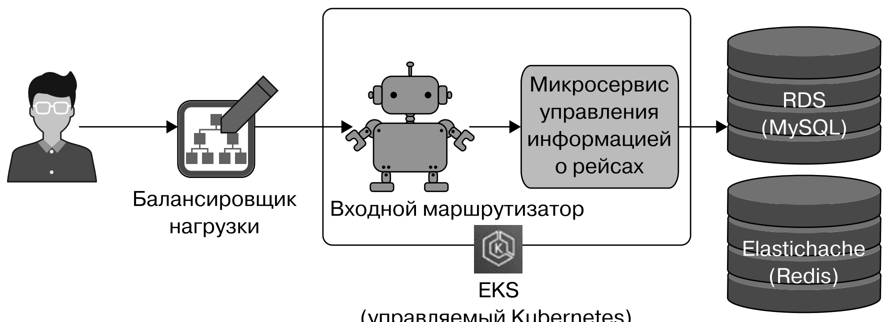
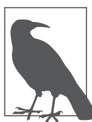


Рис. 10.1. Развертывание в обкатку



Из-за объема сведений, которые нам нужно охватить, мы развернем только микросервис, поставляющий информацию о рейсах. Однако вы можете использовать те же механизмы, что описаны здесь, для развертывания сервиса бронирования.

Для решения поставленной задачи мы используем три разных репозитория GitHub с их собственными конвейерами и ресурсами (как показано на рис. 10.2). Одна из причин, почему мы выбрали такой подход, заключается в том, что он хорошо согласуется с операционной моделью, которую мы определили в главе 2, наделяет каждую из наших команд собственными обязанностями и предоставляет области для работы.

Рис. 10.2. Три репозитория кода для развертывания

Нам предстоит многое обсудить, поэтому перейдем к первому шагу: подготовке среды обкатки на базе AWS.

Настройка среды обкатки

До сих пор мы развертывали микросервисы в локальной среде разработчика. Теперь мы возьмем сервисы, которые создали и протестировали локально, и развернем их в облачной инфраструктуре на базе AWS. В этом разделе мы создадим инфраструктуру обкатки, используя процесс, показанный на рис. 10.3.

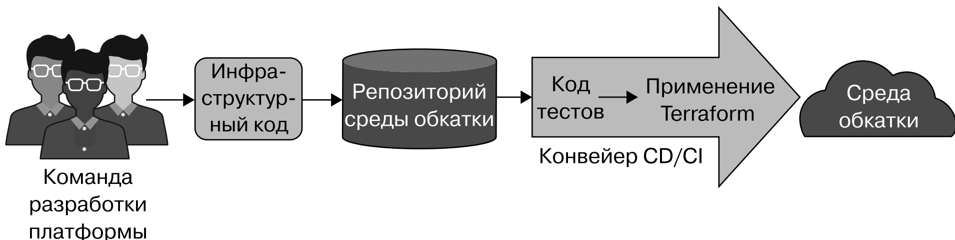


Рис. 10.3. Создание среды обкатки

Мы начали эту работу, когда создавали среду «песочницы» в главе 7. Теперь нам нужно обновить код инфраструктуры, чтобы отразить потребности микросервисов с информацией о рейсах и бронирования авиабилетов. Мы добавим три новых компонента в код Terraform для поддержки микросервисов:

- входной контроллер и пограничный маршрутизатор, пересылающий запросы микросервисам в Kubernetes;
- экземпляр базы данных MySQL на основе AWS для микросервиса управления информацией о рейсах;
- экземпляр базы данных Redis на основе AWS для микросервиса бронирования.

Такие нетривиальные изменения могут быть рискованными. Но именно здесь наш выбор неизменяемой инфраструктуры и инфраструктуры как кода (IaC) начинает приносить свои плоды! Мы точно знаем, как производится сборка среды, потому что весь процесс зафиксирован в коде Terraform. Все, что сейчас требуется, — создать модули для каждого из новых компонентов, обновить определение среды и запустить сборку через конвейер CI/CD.

В главе 7 мы подробно рассмотрели процесс написания каждого модуля Terraform, поэтому сейчас мы не будем углубляться в рассуждения, а просто используем ресурсы кода и конфигурации, которые мы написали для вас. Вам останется лишь настроить их в соответствии с вашими потребностями.

Приступим к краткому обзору новых модулей, которые мы будем использовать для подготовки новых компонентов. Начнем с модуля входного шлюза.

Модуль входного шлюза

В главе 9 мы использовали пограничный маршрутизатор Traefik для маршрутизации сообщений в контейнеры с микросервисами. Аналогичную архитектуру мы реализуем в инфраструктуре на базе AWS. Существует множество инструментов, позволяющих выполнять входную маршрутизацию. Например, многие практикующие специалисты используют контроллер входа Nginx (https://oreil.ly/QyHmJ). Traefik — тоже прекрасный вариант, а поскольку мы уже использовали его на стадии разработки, то принимаем решение внедрить его в качестве контроллера и в среде AWS.

КЛЮЧЕВОЕ РЕШЕНИЕ: ВНЕДРИТЬ TRAEFIK В КАЧЕСТВЕ ВХОДНОГО КОНТРОЛЛЕРА

Мы используем Traefik для маршрутизации сообщений от балансировщика нагрузки к микросервисам, развернутым в Kubernetes.

В целях экономии времени мы написали модуль Terraform заранее, он установит контроллер входа Traefik в среду. Мы сможем использовать этот модуль в коде Terraform среды так же, как сетевые модули, EKS и Argo CD, которые мы использовали в главе 6. Код для модуля Traefik доступен на странице GitHub: https://oreil.ly/8YXIW.

У нас не будет времени на реализацию BFF API в нашем примере. Но входной шлюз, который мы настраиваем, прекрасно подойдет для этой цели в будущем. Например, вы сможете настроить шлюз AWS API Gateway (https://oreil.ly/KATjx) перед входным контроллером, чтобы объединить сервисы в общий API. Фактически мы внедрили Traefik с помощью AWS Network Load Balancer, что упрощает реализацию такого рода подключений.

У нас будет возможность использовать этот входной модуль в подразделе «Разветвление проекта инфраструктуры обкатки» далее, когда мы создадим среду обкатки. А пока рассмотрим вопросы поддержки потребностей базы данных.

Модуль базы данных

Наши микросервисы используют разные базы данных, поэтому мы должны подготовить две базы данных в среде инфраструктуры. Для поддержки потребностей наших команд разработки микросервисов нам понадобятся MySQL и Redis. Мы решили использовать управляемые AWS версии этих продуктов. Таким образом, наша команда разработки платформы сможет предложить командам разработки микросервисов две базы данных в формате «все как сервис».

КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ ОБЩИЕ И УПРАВЛЯЕМЫЕ СЕРВИСЫ БАЗ ДАННЫХ

Команда разработки платформы создаст модули на основе Terraform для подготовки управляемых баз данных AWS для каждой среды.

В нашем модуле базы данных мы используем сервис AWS ElastiCache для подготовки хранилища Redis и AWS Relational Database (RDS) для подготовки экземпляра MySQL. Мы уже написали модуль, который делает это, вы можете найти его в репозитории GitHub этой книги (https://oreil.ly/Microservices_UpandRunning_mod_awsdb). Модуль подготавливает обе базы данных, а также определяет конфигурацию сети и политики доступа, необходимые сервису баз данных для работы.

Применив модуль к среде обкатки, мы получим готовые к использованию экземпляры баз данных Redis и MySQL.

Нам осталось только задействовать наши модули в файле с кодом Terraform и создать среду. Этот процесс мы рассмотрим в следующем подразделе.

Разветвление проекта инфраструктуры обкатки

Среда обкатки, необходимая для выпуска микросервисов, очень похожа на среду «песочницы», которую мы создали в главе 7. Мы продолжим использовать те же методы и принципы, которые применяли ранее. С помощью Terraform мы определим код настройки среды и используем в нем модули, которые написали для сети, кластера Kubernetes и Argo CD. Добавим в него новые модули базы данных и входного контроллера, которые только что описали. И наконец, используем конвейер GitHub Actions для подготовки среды, как сделали это для среды «песочницы».

В главе 6 мы объяснили, как создать конвейер GitHub Actions, а в главе 7 описали процесс написания и применения кода Terraform, поэтому не будем повторяться, а просто используем скелет проекта среды обкатки, который мы уже создали для вас (рис. 10.4). Вам нужно будет внести несколько небольших изменений в код, чтобы он работал в вашей среде AWS. Для этого мы разветвим репозиторий, чтобы у вас была своя копия, которую можно изменять по своему усмотрению.

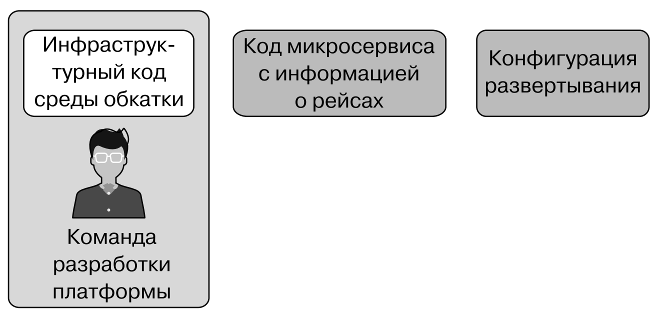


Рис. 10.4. Начало работы с репозиторием среды обкатки

В GitHub *разветвление* (fork) позволяет создать копию чужого проекта с кодом в вашей учетной записи. Чтобы разветвить репозиторий среды обкатки, выполните следующие действия.

1. Откройте браузер и войдите в свою учетную запись GitHub.
2. Перейдите к репозиторию GitHub книги (https://oreil.ly/Microservices_Upand-Running_infrastaging_env).
3. Нажмите кнопку Fork (Разветвление) в правом верхнем углу экрана.

Вы можете дублировать данный репозиторий вместо разветвления. Это позволит изменить режим доступа к репозиторию на закрытый вместо общедоступного. Инструкции по дублированию репозитория GitHub доступны в документации GitHub (https://oreil.ly/HbZMN).

Как только операция будет завершена, у вас появится своя ответвленная копия `infra-stagingenv`. Но она еще не настроена для использования вашей учетной записи или ресурсов AWS. Первое, что вам нужно обновить, — это рабочий поток задач GitHub Actions.

Настройка потока среды обкатки

Поток задач CI/CD, который мы только что получили в результате ветвления, не сможет получить доступ к вашей учетной записи AWS без учетных данных. Поэтому вам нужно добавить в секреты репозитория учетные данные управления доступом к AWS и пароль MySQL. У вас должны быть данные операционной учетной записи в AWS, полученные при настройке конвейера, которую мы выполнили в главе 6. Если у вас нет этих ключей, то можно открыть консоль управления AWS в браузере и создать новый набор учетных данных для операционного пользователя.

Восстановив учетные данные, перейдите на панель Settings (Настройки) ответвленного репозитория GitHub и выберите пункт Secrets (Секреты) в меню навигации слева. Добавьте секреты, перечисленные в табл. 10.1, нажав кнопку New secret (Создать секрет).

Таблица 10.1. Секреты инфраструктуры

Ключ	Значение
AWS_ACCESS_KEY_ID	Идентификатор ключа доступа для вашего операционного пользователя в AWS
AWS_SECRET_ACCESS_KEY	Секретный ключ для вашего операционного пользователя
MYSQL_PASSWORD	microservices

Имена ключей должны вводиться в точности, как показано в табл. 10.1. Если вы допустите ошибку, то конвейер не сможет получить доступ к вашему экземпляру AWS и создавать ресурсы. Используйте значение `microservices` для секрета `MYSQL_PASSWORD`. Этот пароль будет использоваться при подготовке базы данных AWS RDS.

Когда мы разветвляли репозиторий infra-staging-env, GitHub создал копию потока задач Actions, определяющего конвейер CI/CD. Но по соображениям безопасности GitHub не включает автоматически функцию GitHub Actions при разветвлении репозитория (рис. 10.5). А значит, вам нужно запустить ее, выполнив следующие действия:

- перейдите на вкладку Actions (Действия) в консоли управления для вашего ответвленного репозитория;
- если возникнут проблемы, то дайте GitHub указание включить поток задач, который мы ответвили.

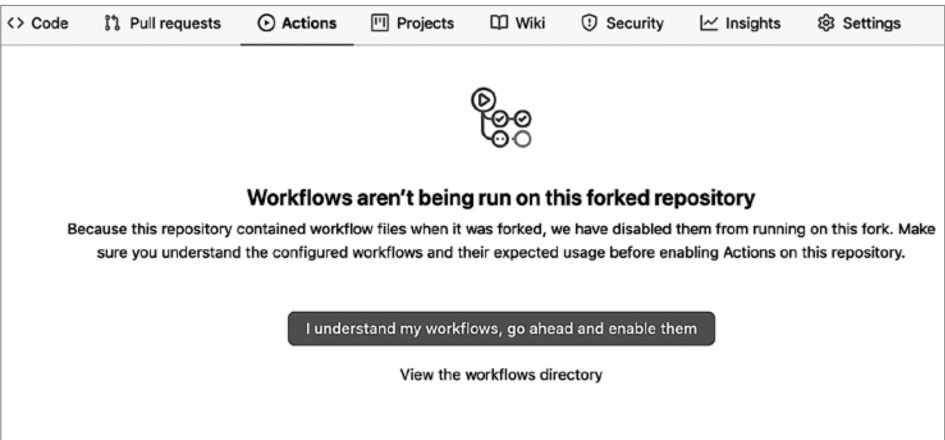
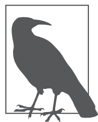
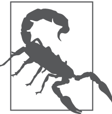


Рис. 10.5. Включение GitHub Actions



GitHub довольно часто меняет свой пользовательский интерфейс, поэтому конкретные шаги и экранные формы могут отличаться от представленных в книге.

Ответвленный инфраструктурный конвейер теперь активирован и готов к запуску и нам осталось лишь внести несколько изменений в код Terraform, который создает среду обкатки.



Поток задач среды обкатки, который мы создали для вас, автоматически сгенерирует файл kubeconfig. Этот файл содержит информацию о подключении, чтобы у вас была возможность подключиться к кластеру Kubernetes, который мы создадим на EKS. Поскольку этот репозиторий кода общедоступен, данный файл будет доступен любому, кто посетит ваш репозиторий. Теоретически это не должно создать проблем. Нашему кластеру EKS требуются учетные данные AWS для аутентификации и подключения. Это означает, что даже с помощью файла kubeconfig злоумышленник не сможет подключиться к вашему кластеру, если у него нет ваших учетных данных AWS.

Редактирование кода инфраструктуры обкатки

Написанный нами код Terraform подготовит среду обкатки. Но он не будет работать, если не установить некоторые значения локальных переменных, характерные для вашей учетной записи AWS и среды. Для этого мы поработаем с файлами в вашей локальной среде. Кроме того, вам нужно будет создать клон ответвленного репозитория infra-staging-env. Оставим это вам как самостоятельное упражнение.

Если вы не знаете точно, как клонировать свой репозиторий, то следуйте инструкциям для вашей ОС в документации GitHub (https://oreil.ly/hvEWn).

Мы отредактируем файл `main.tf`, определяющий среду обкатки. Вам понадобится изменить значения нескольких локальных переменных, которые мы перечислили в табл. 10.2.

Таблица 10.2. Значения переменных для среды обкатки в файле main.tf

Ресурс	Название свойства	Описание
terraform	bucket	Имя корзины S3 для серверного хранилища Terraform
terraform	key	Идентификатор, который будет использоваться для доступа к данным в корзине S3
terraform	region	Ваш регион AWS
locals	aws_region	Ваш регион AWS

Эти значения будут идентичны конфигурации, созданной в главе 7, поэтому если тот код у вас под рукой, то можете скопировать эти значения оттуда.

Чтобы внести эти изменения, отредактируйте файл `main.tf` в текстовом редакторе и подставьте соответствующие значения. Все значения, которые нужно заменить, находятся в разделах `terraform` и `locals` в начале файла. Вы также можете воспользоваться моментом, чтобы просмотреть файл Terraform и узнать больше о том, что он делает. Когда закончите, начало вашего файла должно выглядеть так же, как в примере 10.1.

Пример 10.1. Обновленный файл main.tf для среды обкатки

```
terraform {
  backend "s3" {
  bucket = "rm-terraform-backend"
  key = "terraform-env"
  region = "eu-west-2"
  }
}

locals {
  env_name = "staging"
  aws_region = "eu-west-2"
  k8s_cluster_name = "ms-cluster"
}
```

Теперь код Terraform, определяющий среду обкатки, готов к применению. Но, если попытаться его использовать, он не сможет решить свою задачу, потому что наша операционная учетная запись в AWS не имеет нужных привилегий. Мы добавили модули создания баз данных, но используемая учетная запись не имеет права создавать эти ресурсы AWS или работать с ними. Если попытаться запустить наш код Terraform прямо сейчас, то мы получим ошибки доступа от AWS.

Чтобы решить эту проблему, нужно предоставить операционной учетной записи AWS еще несколько разрешений.

Мы сделаем это, создав новую группу IAM для работы с базой данных, а затем добавим в нее нашу операционную учетную запись, чтобы она унаследовала разрешения группы.

Выполните следующую команду AWS CLI, чтобы создать новую группу `DB-Ops`:

```
$ aws iam create-group --group-name DB-Ops
```

Далее подключите к группе политики доступа для RDS и ElastiCache:

```
$ aws iam attach-group-policy --group-name DB-Ops\
 --policy-arn arn:aws:iam::aws:policy/AmazonRDSFullAccess &&\
aws iam attach-group-policy --group-name DB-Ops\
 --policy-arn arn:aws:iam::aws:policy/AmazonElastiCacheFullAccess
```

Наконец, добавьте нашу операционную учетную запись в только что созданную группу:

```
$ aws iam add-user-to-group --user-name ops-account --group-name DB-Ops
```

С установленными разрешениями мы почти готовы к работе. Но перед отправкой изменений в репозиторий давайте быстро протестируем обновленный код инфраструктуры. Выполните следующие команды Terraform, чтобы отформатировать и проверить наш обновленный код:

```
infra-staging-env$ terraform fmt
[...]
infra-staging-env$ terraform init
[...]
infra-staging-env$ terraform validate
[...]
infra-staging-env$ terraform plan
[...]
```

Теперь мы готовы зафиксировать код инфраструктуры и запустить конвейер CI/CD. Начнем с фиксации обновленного кода Terraform в ответвленном репозитории:

```
$ git add .
$ git commit -m "Staging environment with databases"
$ git push origin
```

Как вы наверняка помните, в главе 6 мы настроили запуск нашего потока задач при отправке тега выпуска, начинающегося с `v`. Используйте следующие команды Git, чтобы создать новый тег `v1.0` и поместить его в свой ответвленный репозиторий:

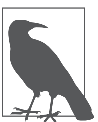
```
$ git tag -a v1.0 -m "Initial staging environment build"
$ git push origin v1.0
```

После этого процесс инициализации среды обкатки должен запуститься. Состояние конвейера можно проверить в веб-консоли GitHub. Если задание конвейера выполнится успешно, то вы получите среду обкатки с кластером Kubernetes и готовыми к использованию базами данных MySQL и Redis. Этот

кластер Kubernetes понадобится нам для развертывания микросервисов, поэтому следующим шагом проверим его готовность к работе.

Проверка доступности кластера Kubernetes

Чтобы взаимодействовать с кластером обкатки Kubernetes, нам понадобятся сведения о конфигурации для приложения `kubectl`. Мы получим их, использовав тот же прием, что и в подразделе «Тестирование среды» в главе 7, — загрузим файл конфигурации и обновим локальные настройки среды.



Убедитесь, что создание конвейера CI/CD успешно завершено, прежде чем пытаться подключиться к кластеру Kubernetes.

Настройте среду клиента Kubernetes, скачав файл `kubeconfig`, созданный конвейером среды обкатки GitHub Actions и сохранив путь к нему в переменной среды `KUBECONFIG`:

```
$ export KUBECONFIG=~/Downloads/kubeconfig
```

Затем выполните команду `kubectl get svc --all-namespaces`, чтобы подтвердить, что кластер среды обкатки запущен и объекты Kubernetes развернуты. Вы должны получить результат, который выглядит аналогично представленному в примере 10.2.

Пример 10.2. Результат get svc

```
$ kubectl get svc --all-namespaces

NAMESPACE   NAME                              TYPE         CLUSTER-IP
argocd      msur-argocd-application-controller ClusterIP    172.20.133.240
argocd      msur-argocd-dex-server            ClusterIP    172.20.74.68
default     ms-ingress-nginx-ingress          LoadBalancer 172.20.239.114
[... множество других сервисов ...]
```

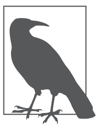
В результате вы должны увидеть список всех сервисов Kubernetes, которые мы развернули, включая сервисы для приложения Argo CD и сервис контроллера входа Nginx. Это означает, что кластер запущен и в нем развернуты необходимые нам сервисы.

Создание секретов в Kubernetes

Последний шаг, который мы должны сделать, — настроить объект Secret в Kubernetes. Для подключения к MySQL нашим микросервисам, поставляющим информацию о рейсах, потребуется пароль. Чтобы не хранить его в открытом виде, мы используем специальный объект Kubernetes, скрывающий данные от неавторизованных пользователей.

Выполните следующую команду, чтобы создать объект Secret в Kubernetes и сохранить в нем пароль MySQL:

```
$ kubectl create secret generic \
mysql --from-literal=password=microservices -n microservices
```

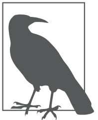


Встроенные в Kubernetes средства хранения секретов полезны, но мы рекомендуем использовать нечто более многофункциональное для надлежащей реализации. В этой области доступно множество вариантов, в том числе HashiCorp Vault (https://oreil.ly/YeiHQ).

Теперь у нас есть среда обкатки с инфраструктурой, соответствующей потребностям разработанных нами микросервисов. Следующий шаг — публикация этих микросервисов в виде контейнеров, чтобы мы могли развернуть их в среде.

Отправка контейнера с информацией о рейсах

В главе 9 мы использовали `make` для тестирования, сборки и запуска микросервисов в локальной среде разработки. Но для создания и развертывания сервисов в средах тестирования, обкатки и т. д. нужен более повторяемый и автоматизированный процесс.



Мы уже создали контейнерную версию микросервиса управления информацией о рейсах (https://oreil.ly/7LHnY), чтобы вы могли ею воспользоваться. Поэтому, если вы не собираетесь самостоятельно создавать поток задач развертывания Docker Hub, то можете перейти к разделу «Развертывание контейнера с информацией о рейсах» далее в этой главе.

Использование методов автоматизации и DevOps для создания сервисов повышает предсказуемость, качество и скорость развертывания микросервисов. Это

тот же принцип, который мы применили к созданию инфраструктуры. В данном разделе мы построим конвейер непрерывной интеграции и непрерывной доставки (CI/CD) для создания и публикации микросервиса управления информацией о рейсах в реестре контейнеров, как показано на рис. 10.6.

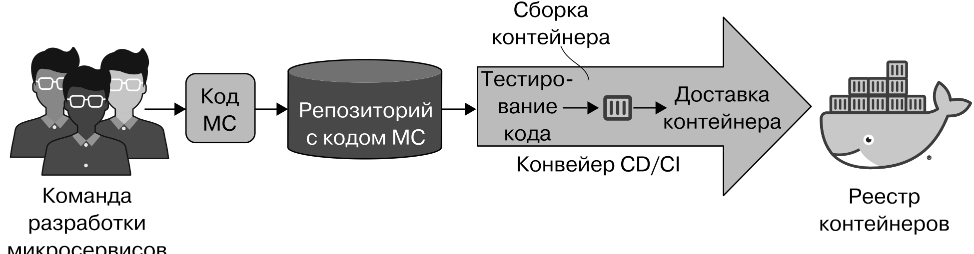


Рис. 10.6. Конвейер CI/CD микросервисов

Начнем с обзора Docker Hub, реестра контейнеров, который мы используем для размещения контейнеров микросервисов.

Введение в Docker Hub

В главе 9 в процессе сборки на основе `make` мы использовали `docker-compose` и `docker` для создания контейнеров тестирования и выпуска. Чтобы поместить эти контейнеры в среду обкатки, понадобится некоторый способ их перемещения или *отправки*. Контейнеры во многом похожи на двоичные приложения, поэтому мы могли бы просто выгрузить их в файловую систему целевой среды. Но это может привести к беспорядку с увеличением количества контейнеров.

Вместо этого отправим контейнеры в *реестр контейнеров* — программную систему хранения контейнеров. Хороший реестр обеспечивает безопасность контейнеров и позволяет отыскивать их, обновлять и изменять. Docker даже определяет API (https://oreil.ly/sLI3B) для операций с реестром и имеет встроенную поддержку для его использования.

Существует множество доступных вариантов размещения реестра, поддерживающих Docker API. Все основные облачные провайдеры могут разместить защищенный закрытый реестр. Вы также можете настроить свой сервер реестра, используя реализацию Docker с открытым исходным кодом. Для примеров в этой книге мы задействуем общедоступный реестр Docker под названием Docker Hub. Мы выбрали Docker Hub, потому что он бесплатный, популярный и хорошо интегрируется с GitHub Actions.

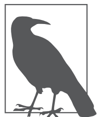
КЛЮЧЕВОЕ РЕШЕНИЕ: ИСПОЛЬЗОВАТЬ DOCKER HUB В КАЧЕСТВЕ РЕЕСТРА КОНТЕЙНЕРОВ

Мы отправим наш контейнер с микросервисом в реестр Docker Hub.

Настройка Docker Hub

Настроить новый реестр Docker Hub довольно просто — достаточно войти в Docker Hub и создать реестр. Чтобы создать реестр для нашего приложения управления бронированием, выполните следующие действия.

1. Откройте в браузере главную страницу Docker Hub (https://hub.docker.com).
2. Войдите в учетную запись Docker Hub.
3. Нажмите кнопку Create Repository (Создать репозиторий).
4. Присвойте репозиторию название flights.
5. Нажмите кнопку Create (Создать).



Чтобы использовать Docker Hub, необходимо иметь учетную запись Docker. Если вы установили Docker при настройке среды разработчика, то у вас уже есть идентификатор Docker. Если у вас его нет, то перейдите на https://hub.docker.com и создайте учетную запись.

Если в ходе этого процесса возникнут проблемы, то загляните в документацию Docker (https://oreil.ly/owCnP). Имея учетную запись Docker и репозиторий контейнеров, мы готовы создавать и выгружать в него контейнеры с помощью конвейера CI/CD.

Настройка конвейера

До сих пор мы использовали GitHub Actions для реализации конвейера и выполнения работ по подготовке инфраструктуры на основе подхода IaC. Он прекрасно справляется со своими задачами, поэтому для согласованности мы снова применим Actions в роли конвейера для сборки контейнеров микросервисов. Как дополнительное преимущество мы сможем воспользоваться действиями, опубликованными Docker, упрощающими интеграцию потока задач в Docker Hub.

В главе 9 мы рассмотрели процесс создания микросервиса управления информацией о рейсах. Если вы выполнили эти шаги, то у вас должен быть репозиторий GitHub, содержащий код и файлы makefile (рис. 10.7). Мы создадим поток

задач GitHub Actions внутри репозитория, чтобы конвейер CI мог находиться рядом с кодом. Если у вас еще нет своего репозитория для сервиса управления информацией о рейсах, то создайте ответвление примера репозитория этой книги (https://oreil.ly/Microservices_UpandRunning_msflights).

Рис. 10.7. Создание контейнера в репозитории сервиса рейсов

Как и раньше, начнем настройку конвейера с добавления учетных данных в репозиторий GitHub.

Настройка секретов Docker Hub

Для публикации контейнера поток задач должен взаимодействовать с Docker Hub. Поэтому необходимо добавить в репозиторий микросервиса ms-flights секреты с информацией о доступе к Docker. В частности, нужно добавить два секрета, перечисленные в табл. 10.3. Это те же учетные данные, которые вы использовали бы для входа в Docker Hub.

Таблица 10.3. Секреты GitHub для доступа к Docker Hub

Ключ	Описание
DOCKER_USERNAME	Идентификатор учетной записи Docker
DOCKER_PASSWORD	Пароль учетной записи Docker

Мы уже несколько раз обсуждали детали настройки секретов, но все равно напомним, что вам нужно будет выполнить следующее.

1. Открыть в браузере страницу настроек вашего ответвленного репозитория ms-flights в GitHub.

2. Выбрать пункт Secrets (Секреты) на боковой панели навигации.
3. Добавить требуемый секрет, нажав кнопку New Secret (Создать секрет).

Возможно, вы обратили внимание, что мы не добавляем никаких секретов учетной записи AWS в этот репозиторий. Причина проста: этот конвейер не будет развертывать экземпляр AWS. Данный поток задач будет сосредоточен только на размещении контейнеров в реестре Docker Hub, а не на развертывании контейнеров в среде обкатки.

Такое разделение помогает сделать контейнеры микросервисов переносимыми и не зависящими от среды (это означает, что мы не будем добавлять в сборку какую-либо логику или значения, специфичные для той или иной среды). Использование одного и того же контейнера во всех средах тестирования и выпуска должно повысить надежность системы в целом.

Все, что нам нужно сделать сейчас, — это создать поток задач, который выполняет работу по сборке, тестированию и отправке контейнера.

Отправка контейнера сервиса с информацией о рейсах

Если вы ответвили репозиторий ms-flights, то обнаружите, что мы уже написали поток задач GitHub Actions, который создает и отправляет контейнер. Вам остается только включить его на вкладке Actions (Действия) в своем ответвленном репозитории, где вам будет предложено активировать поток задач. Если вы создали свой репозиторий ms-flights, то скопируйте код потока задач (https://oreil.ly/Microservices_UpandRunning_mainyaml) в собственный каталог потоков задач.

Поток задач GitHub Actions, который мы определили, запускается тегом выпуска и содержит такие шаги.

1. Запускает модульные тесты для кода.
2. Создает контейнер с микросервисом.
3. Помещает контейнер в реестр Docker Hub.

Мы уже добавили учетные данные Docker Hub в репозиторий, поэтому нам нужно лишь отправить тег v1.0 в выпуск, чтобы запустить поток задач CI/CD. Мы уже делали это в локальной копии репозитория с помощью командной строки. Но теперь для экономии времени запустим сборку из веб-интерфейса GitHub.

В браузере перейдите на вкладку Code (Код) вашего ответвленного репозитория ms-flights на GitHub. Щелкните на ссылке Create a new release (Создать

новый релиз) в разделе Releases (Релизы) справа на странице, как показано на рис. 10.8.

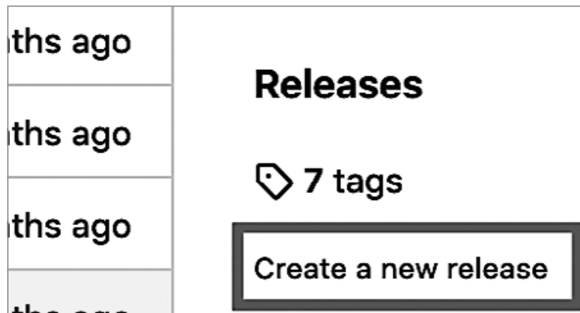


Рис. 10.8. Создание нового релиза

Затем введите значение `v1.0` в поле версии тега, как показано на рис. 10.9. Затем нажмите кнопку Publish Release (Опубликовать релиз) в нижней части экрана.

Рис. 10.9. Установка версии тега

Публикация релиза GitHub с тегом `v1.0` — то же самое, что вставить тег с помощью Git CLI. Конечным результатом должен стать запуск нашего потока задач Actions на GitHub. Вы можете перейти на вкладку Actions (Действия) в репозитории и убедиться, что поток задач `CICD` запущен. Запуск файла makefile и упаковка контейнера займет несколько минут. Когда это будет сделано, контейнер сервиса с информацией о рейсах будет отправлен в реестр Docker Hub и готов к использованию.

Чтобы убедиться, что контейнер оправлен, зайдите в учетную запись Docker Hub в браузере и просмотрите свои репозитории. Вы должны увидеть запись для только что отправленного контейнера. Она будет выглядеть примерно так, как показано на рис. 10.10.

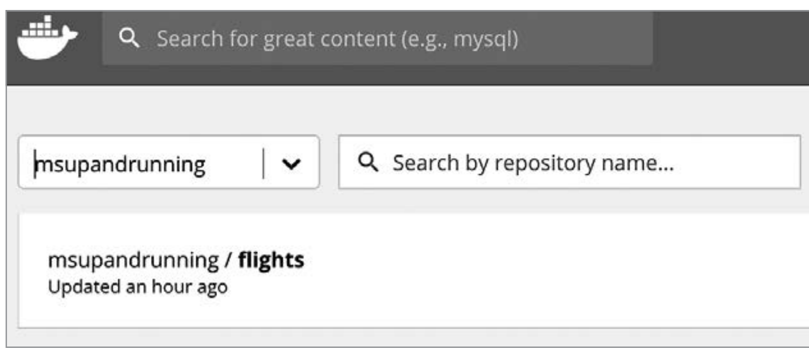


Рис. 10.10. Контейнер ms-flights передан

Теперь у нас есть контейнер с микросервисом ms-flights, готовый к развертыванию в среде обкатки. Поместив контейнеры с микросервисами в реестр, можно переходить к их развертыванию.

Развертывание контейнера с информацией о рейсах

Теперь у нас есть все необходимое для развертывания микросервиса управления информацией о рейсах. Мы подготовили тестовую среду с использованием конвейера инфраструктуры и создали образ контейнера для развертывания сервиса. Чтобы завершить работу по развертыванию, мы используем инструмент Argo CD GitOps, который установили в стек инфраструктуры в главе 7. К концу этого раздела мы получим развернутую и готовую к использованию версию микросервиса управления информацией о рейсах.

Чтобы упростить многократное развертывание, создадим новый репозиторий развертывания, где будут храниться пакеты Helm. Мы представили Helm в подразделе «Настройка Argo CD» в главе 7, когда создавали первую среду. Создаваемые пакеты Helm будут описывать, как развертывать микросервис, и использоваться приложением Argo CD для развертывания контейнеров в среде обкатки (рис. 10.11).

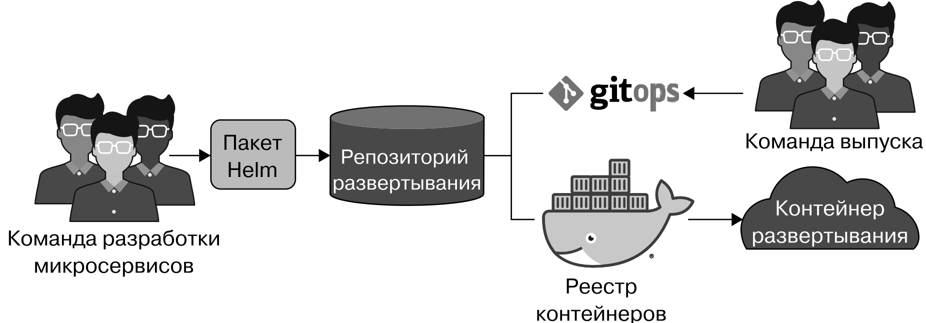


Рис. 10.11. Пакет Helm будет определять развертывание сервиса

Вся работа по развертыванию будет происходить в мире Kubernetes, поэтому рассмотрим в общих чертах, что это значит.

Особенности развертывания в Kubernetes

В главе 7 мы представили и установили Kubernetes, чтобы помочь себе в запуске и эксплуатации контейнерных микросервисов. Kubernetes пользуется заслуженной популярностью, выполняя бо́льшую часть работы для запуска контейнеров, проверки их работоспособности, поиска сервисов, их репликации и повторного запуска при сбое. Он придает системе устойчивость и способность к самовосстановлению, которые помогут соответствовать нашим руководящим принципам.

Но кластер Kubernetes все равно нуждается в указаниях, что делать. Он не может развернуть ваши микросервисы, не зная, где искать образы контейнеров. Он не может проверить работоспособность микросервиса, не зная, какой API вызывать. Также для кластера Kubernetes необходимо определить ограничения на количества экземпляров контейнеров и как к этим сервисам следует обращаться по сети.

Kubernetes видит мир как набор декларативных конфигурационных объектов. Чтобы настроить развертывание микросервисов, нужно описать оптимальное состояние запущенного контейнера. Если вы правильно опишете свою конфигурацию, то Kubernetes приложит все силы, чтобы воплотить ваш сервис в жизнь — и сохранить его таким.

Этот декларативный подход подобен тому, что мы применяли для описания ресурсов инфраструктуры с помощью Terraform. В Kubernetes мы определим набор специальных объектов развертывания, используя формат YAML. Фреймворк Kubernetes невероятно сложен, поэтому мы не сможем подробно описать его на

страницах этой книги, но расскажем о некоторых основных его объектах, чтобы вы могли понять, как будут развертываться микросервисы.

Объекты и контроллеры Kubernetes

Для полноценной работы с платформой Kubernetes необходимо знать и понимать особенности многих объектов. Но для наших целей достаточно поверхностного знакомства с пятью ключевыми объектами, необходимыми для создания простого пакета развертывания микросервиса управления информацией о рейсах: `Pod`, `ReplicaSet`, `Deployment`, `Service` и `Ingress`.

- Объект `Pod` описывает базовую единицу рабочей нагрузки. Он определяет один или несколько контейнеров Docker, которые должны запускаться и поддерживаться вместе.
- Объекты `ReplicaSet` сообщают Kubernetes, сколько экземпляров объекта `Pod` должно выполняться одновременно. Обычно работать с объектами `ReplicaSet` напрямую не требуется.
- Контроллер `Deployment` объявляет желаемое состояние для `Pod` и связанных с ним `ReplicaSet`. Это основной объект, необходимый для развертывания микросервисов в Kubernetes.
- Объект `Service` определяет, как приложения в кластере Kubernetes могут получить доступ к `Pod` по сети, даже если одновременно запущено несколько точных копий. Он позволяет определить единый IP-адрес и порт для доступа к группе модулей Pod. Разработчики почти всегда определяют объект `Service` для развертывания микросервиса.
- Объект `Ingress` определяет маршрут входа в `Service` для приложений за пределами кластера. Объявление `Ingress` может включать правила маршрутизации, чтобы контроллер входа мог передавать сообщения правильным объектам `Service`.

Чтобы развернуть микросервис, мы должны написать декларативные конфигурации объектов `Ingress`, `Service` и `Deployment`. Однако мы не будем определять конфигурации `Pod` и `ReplicaSet` в отдельных файлах, а включим их детали в конфигурацию объекта `Deployment`. Как упоминалось ранее, будем использовать Helm для упаковки всех этих файлов.

Создание чарта Helm

Контроллер `Deployment` в Kubernetes может потребовать тесного взаимодействия с кластером. Вам нужно выполнить несколько вызовов Kubernetes API, чтобы сообщить ему, как, когда и где развернуть контейнеры. Для преодо-

ления некоторых из этих сложностей мы используем инструмент упаковки Helm.

Helm — это диспетчер пакетов для Kubernetes. Он предлагает более простой способ управления установкой и развертыванием приложения в кластере Kubernetes. Ранее в книге мы использовали Helm для установки готовых пакетов, таких как Argo CD. Теперь мы напишем свой пакет Helm, чтобы так же легко устанавливать свои микросервисы.

Для работы с Helm важно знать и понимать три основных понятия: чарты, шаблоны и значения.

- *Чарты* — наборы файлов, описывающих ресурс или развертывание Kubernetes. Чарт — основная единица развертывания в Helm. Ранее в этой книге мы использовали готовые чарты, когда развертывали приложения на базе Kubernetes, такие как Argo CD.
- *Шаблоны* — файлы в чарте, описывающие ресурсы Kubernetes. Они называются шаблонами, потому что содержат специальные инструкции, которые Helm использует для замены значений в файле. Например, вы можете создать шаблон `Service` для микросервиса и сделать номер порта `Service` шаблонным значением.
- *Значения*. Каждый чарт имеет файл значений для заполнения шаблона. Файлы значений — удобный способ управления различиями между средами. Значения также можно переопределять при установке чарта Helm.

Чтобы получить Helm-пакет для микросервиса управления информацией о рейсах, мы должны создать чарт Helm. В его рамках мы определим набор файлов шаблонов, описывающих, как должен развертываться сервис. Наш шаблон будет иметь некоторые параметризованные значения, которые сделают его пригодным для использования в разных средах. Наконец, мы создадим файл значений для среды обкатки, который заполняет наши шаблоны.

Как было описано в начале данного раздела, нам нужно, чтобы чарты Helm были доступны приложению Argo CD. Поэтому первым делом нам нужно создать репозиторий развертывания микросервисов для их хранения и управления ими.

Создание репозитория развертывания микросервисов

Мы будем хранить чарты Helm в едином монорепозитории развертываний микросервисов. Это хорошо вписывается в операционную модель и позволяет команде выпуска единообразно управлять фактическим выпуском сервисов. Команды разработки микросервисов по-прежнему могут иметь собственные

чарты развертывания Helm и независимо развертывать их в репозитории развертывания (рис. 10.12).

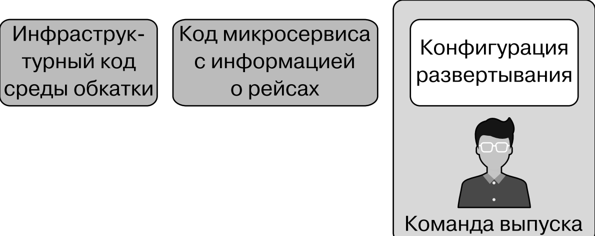


Рис. 10.12. Создание пакета развертывания в репозитории развертывания

Для начала создайте новый репозиторий GitHub под названием ms-deploy. Затем создайте локальный клон репозитория в своей среде разработки. Мы уже делали это несколько раз, поэтому не будем снова вдаваться во все подробности. Если вам нужно освежить в памяти процесс создания репозитория, то мы рекомендуем обратиться к документации GitHub Quickstart (https://oreil.ly/TdrGG).

Репозиторий развертывания, который вы сейчас создаете, станет «источником истины» для инструмента развертывания Argo CD GitOps. Его мы настроим позже в этой главе.

Теперь у вас должен быть пустой репозиторий Git, готовый к заполнению пакетами Helm.

Самый простой способ начать работу с файлами Helm — использовать приложение Helm CLI. Его интерфейс позволяет создавать, устанавливать и проверять чарты Helm с помощью Kubernetes API. В своих примерах мы будем использовать Helm версии 3.2.4, который можно найти на сайте GitHub: https://oreil.ly/ohMF7.

Если у вас еще нет Helm CLI, то скачайте и установите его на свой локальный компьютер прямо сейчас. После этого вы будете готовы создать чарт Helm для ms-flights.

Создание чарта Helm

Одна из приятных особенностей интерфейса Helm CLI — удобная функция для быстрого создания нового чарта. Итак, чтобы создать заготовку чарта,

перейдите в корневой каталог в репозитории ms-deploy и выполните следующую команду:

```
ms-deploy $ helm create ms-flights
```

После этого Helm создаст базовый пакет, содержащий файл `chart.yaml`, который описывает чарт, файл `values.yaml`, который можно использовать для настройки значений в чарте, и каталог `templates` с целым набором шаблонов YAML для развертывания в Kubernetes.

Самое замечательное в использовании Helm заключается в том, что он создает бо́льшую часть шаблонного кода, который нам пришлось бы написать для развертывания микросервисов в Kubernetes. Нам остается лишь внести несколько небольших изменений в сгенерированные шаблоны, чтобы получить развертываемый пакет.

В частности, нам нужно обновить файл `templates/deployment.yaml`, чтобы сделать его более специфичным для развертываемого контейнера.

Обновление шаблона развертывания рейсов

Файл `/ms-flights/templates/deployment.yaml` содержит описание объекта Kubernetes, который определяет целевое состояние развертываемого модуля `Pod`. Мы уже упоминали, что объекты Kubernetes могут быть довольно сложными. Хорошая новость в том, что файл, сгенерированный диспетчером Helm, содержит множество значений-заполнителей, которые можно оставить как есть. Но нам нужно внести несколько небольших изменений, чтобы адаптировать это развертывание для микросервиса управления информацией о рейсах.

Для начала познакомимся с некоторыми ключевыми свойствами YAML в объекте развертывания:

- `apiVersion` — каждый файл Kubernetes YAML указывает версию именованного Kubernetes API, который использует этот файл;
- `kind` — определяет тип объекта Kubernetes. В данном случае объектом Kubernetes является `Deployment`;
- `spec` — спецификация для объекта Kubernetes — это суть описания;
- `spec.replicas` — количество копий объекта `Pod`, запускаемых этим развертыванием. Kubernetes создаст наборы реплик ReplicaSet на основе этого значения;

- `spec.template` — свойство `template` спецификации `Deployment` является шаблоном для модуля `Pod`, который планируется развернуть. Kubernetes использует этот шаблон для подготовки развертываемых модулей;
- `spec.template.containers` — свойство `containers` шаблона модуля `Pod` определяет образ контейнера и значения среды, которые Kubernetes должен использовать при создании копии модуля.

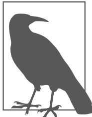
Для нашего простого развертывания мы оставим значения по умолчанию в большинстве свойств объекта `Deployment` и изменим только свойство `spec.template.containers`, чтобы адаптировать его для развертывания созданного нами контейнера ms-flights.

Обновите свойство `containers` в файле YAML, чтобы оно содержало значения `env`, `ports`, `livenessProbe` и `readinessProbe`, как показано в примере 10.3.

Пример 10.3. Спецификация шаблона ms-flights

```
spec:
[...]
  template:
  [...]
    spec:
    [...]
      containers:
        - name: {{ .Chart.Name }}
          [...]
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          env:
            - name: MYSQL_HOST
              value: {{ .Values.MYSQL_HOST | quote }}
            - name: MYSQL_USER
              value: {{ .Values.MYSQL_USER | quote }}
            - name : MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: {{ .Values.MYSQLSecretName }}
                  key: {{ .Values.MYSQLSecretKey }}
            - name: MYSQL_DATABASE
              value: {{ .Values.MYSQL_DATABASE | quote }}
          ports:
            - name: http
              containerPort: 5501
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /ping
              port: http
```

```
readinessProbe:
  httpGet:
    path: /health
    port: http
[...]
```



Завершенный пример чарта Helm для ms-flights доступен в репозитории GitHub книги (https://oreil.ly/Microservices_UpandRunning_msflights).

В раздел `containers` мы добавили следующее:

- набор переменных среды с параметрами подключения к базе данных MySQL (фактические значения установим позже);
- TCP-порт, экспортируемый контейнером, к которому будет привязан микросервис управления информацией о рейсах;
- конечные точки для проверки работоспособности и готовности сервиса со стороны Kubernetes (как определено в главе 9).

Это все, что нужно настроить, чтобы использовать сгенерированные шаблоны Helm в наших интересах. В созданном нами шаблоне развертывания мы определили параметризованный объект Kubernetes `Deployment`.

Теперь осталось определить некоторые значения для использования в шаблоне.

Установка значений пакета

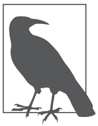
Одна из приятных особенностей применения пакета Helm для развертывания — мы можем повторно использовать один и тот же шаблон для множества разных сред, изменив несколько значений. Задать эти значения можно, например, с помощью клиента Helm во время установки. Мы сделали это ранее в книге, когда устанавливали пакет Helm для Argo CD.

Другой вариант — создать файл, хранящий все необходимые значения в одном месте. Именно такой подход мы используем в своем пакете развертывания. Главное его преимущество — возможность управлять файлами значений развертывания как кодом. Мы будем использовать файл `values.yaml`, сгенерированный диспетчером Helm. Вы найдете его в корневом каталоге чарта ms-flights.

Сначала необходимо обновить сведения об образе Docker. Откройте файл `values.yaml` в текстовом редакторе и найдите ключ `image`. Обновите значение этого ключа, как показано в примере 10.4.

Пример 10.4. Пример образа

```
replicaCount: 1

image:
  repository: "msupandrunning/flights"
  pullPolicy: IfNotPresent
  tag: "v1.0"
```



В этом примере используется контейнер, который мы создали заранее. Если вы решите использовать свой, то измените также значения repository и tag.

Далее нужно добавить значения параметров подключения к MySQL, чтобы микросервис мог подключаться к сервисам баз данных среды обкатки. Добавьте следующий код YAML в файл значений (можете добавить его сразу после свойства `tag`):

```
image:
[..]

MYSQL_HOST: rds.staging.msur-vpc.com
MYSQL_USER: microservices
MYSQL_DATABASE: microservices_db
MYSQLSecretName: mysql
MYSQLSecretKey: password
```

Наконец, найдите свойство `ingress` ближе к концу файла и обновите его, как показано ниже:

```
ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: traefik
  hosts:
    - host: flightsvc.com
      paths: ["/flights"]
```

Это определение настраивает входной шлюз Ingress для пересылки микросервису ms-flights любых сообщений, отправленных на хост flightsvc.com с URI `/flights`. Фактически размещать сервис в домене `flightsvc.com` совсем необязательно, мы лишь должны гарантировать, что HTTP-запросы содержат эти значения, чтобы они достигли нашего сервиса.

Для настройки производственной среды почти наверняка потребуется внести больше изменений в файлы значений и шаблона. Но для обкатки этого более чем достаточно.

Тестирование и фиксация пакета

Наконец нам нужно выполнить пробный прогон и убедиться в отсутствии синтаксических ошибок. Для этого потребуется подключение к кластеру Kubernetes, поэтому удостоверьтесь, что эта среда по-прежнему доступна.

Запустите следующую команду, чтобы проверить — сможет ли Helm собрать пакет:

```
ms-flights$ helm install --debug --dry-run flight-info
```

В случае успеха Helm вернет много кода YAML с определениями объектов, которые он будет генерировать. Код должен завершаться примерно такими строками:

```
[... много кода YAML...]
  backend:
            serviceName: flight-info-ms-flights
            servicePort: 80
NOTES:
1. Get the application URL by running these commands:
  http://flightsvc.com/flights
```

Если у вас возникли проблемы с запуском пакета Helm, то ознакомьтесь с примером пакета в репозитории книги (https://oreil.ly/Microservices_UpandRunning_msflights_ex).

Если вывод не содержит сообщений об ошибках, то отправьте готовые файлы Helm в репозиторий GitHub:

```
ms-flights$ git add .
ms-flights$ git commit -m "initial commit"
ms-flights$ git push origin
```

Теперь, когда файлы пакетов зафиксированы в репозитории развертывания, их можно использовать с помощью инструмента развертывания Argo CD GitOps.

Argo CD для развертывания GitOps

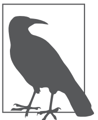
К настоящему моменту мы создали чарт Helm, дающий удобный способ развертывания микросервисов в кластере Kubernetes. Helm поддерживает возможность развертывания в кластерах Kubernetes, поэтому сделанного достаточно, чтобы развернуть сервис с информацией о рейсах в среде обкатки.

Но с текущими настройками практически все операции должны выполняться вручную — нам придется использовать Helm CLI для каждого развертывания. Кроме того, хотелось бы каким-то образом отслеживать текущие состояния и версии развернутых сервисов, чтобы знать, требуется ли новое развертывание при обновлении репозитория развертывания.

Мы можем исправить ситуацию. Ранее в главе 7 мы познакомились с Argo CD — инструментом непрерывного развертывания. Теперь есть возможность использовать его и автоматизировать развертывание сервисов в текущих средах.

Argo CD — инструмент развертывания GitOps, использующий репозиторий Git в качестве источника желаемого состояния развертывания для рабочих нагрузок и сервисов. Проверяя указанный репозиторий, Argo CD определяет, соответствует ли заданное целевое состояние текущему состоянию в среде и при необходимости «синхронизирует» развертывание в соответствии с тем, что мы объявили в чартах Helm.

Этот декларативный подход хорошо согласуется с нашими принципами и другими инструментами, принятыми на вооружение, такими как Terraform. Чтобы все это волшебство заработало, нужно войти в экземпляр Argo CD, установленный в среде обкатки, передать ему ссылку на репозиторий ms-deploy и настроить синхронизированное развертывание.



Не забудьте добавить Kubernetes-объект Secret с паролем MySQL, как описано в разделе «Создание секретов в Kubernetes» выше в этой главе. В противном случае сервис управления информацией о рейсах не запустится.

Авторизация в Argo CD

Прежде чем войти в Argo CD, нужно получить пароль администратора. По умолчанию Argo CD задает пароль, совпадающий с именем объекта Kubernetes, на котором он работает. Выполните следующую команду `kubectl`, чтобы найти `Pod` с Argo CD:

```
$ kubectl get pods -n "argocd" | grep argocd-server
NAME                                READY     STATUS    RESTARTS    AGE
msur-argocd-server-c6d4ffcf-9z4c2   1/1       Running   0           51s
```

Скопируйте куда-нибудь имя модуля, так как оно будет паролем для входа. В частности, в примере выше паролем будет строка `msur-argocd-serverc6d4ffcf-9z4c2`. Чтобы получить доступ к странице входа и ввести свои учетные данные,

нужно настроить правило переадресации портов, потому что изначально мы неправильно определили способ доступа к кластеру Kubernetes из Интернета. К счастью, `kubectl` предоставляет удобный встроенный инструмент для переадресации запросов с локального компьютера в кластер. Выполните следующую команду, чтобы запустить переадресацию:

```
$ kubectl port-forward svc/msur-argocd-server 8443:443 -n "argocd"
Forwarding from 127.0.0.1:8443 -> 8080
Forwarding from [::1]:8443 -> 8080
```

Теперь у вас должна появиться возможность открыть страницу `localhost:8443` в браузере. Вы почти наверняка получите предупреждение о том, что сайту нельзя доверять. Это нормально и ожидаемо на данном этапе. Сообщите своему браузеру, что все в порядке, чтобы продолжить, и затем вы должны увидеть страницу входа, как показано на рис. 10.13.

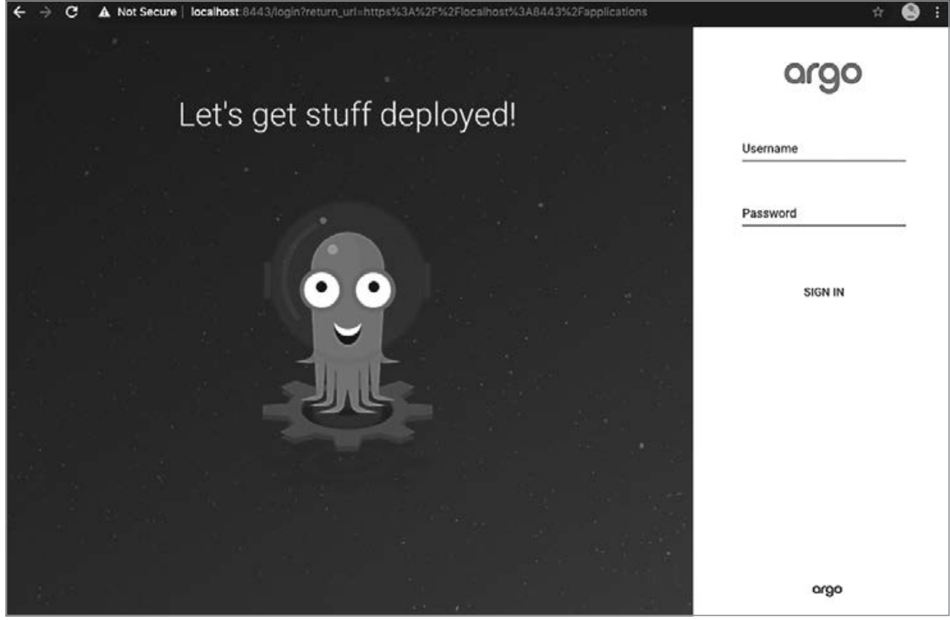


Рис. 10.13. Страница входа в Argo CD

Введите имя пользователя `admin` и пароль, который вы записали ранее, и войдите в систему. В случае успеха вы увидите панель мониторинга. Теперь можно создать ссылку на развертывание сервиса с информацией о рейсах.

Синхронизация и развертывание микросервиса

В Argo CD микросервис или рабочая нагрузка, который необходимо развернуть, называются *приложением*. Чтобы развернуть микросервис с информацией о рейсах, нужно создать новое «приложение» и сконфигурировать его со значениями, которые ссылаются на пакет Helm в репозитории Git, созданном ранее.

Для начала нажмите кнопку Create Application (Создать приложение) или New App (Новое приложение) на панели мониторинга. После этого справа появится веб-форма, которую нужно заполнить метаданными о приложении и местоположении пакета Helm. В нашем случае Argo CD должен извлечь пакет из каталога ms-flights в репозитории развертывания.

Используйте значения из табл. 10.4 для настройки развертывания микросервиса с информацией о рейсах. Обязательно замените значение `YOUR_DEPLOYMENTS_REPOSITORY_URL` на URL репозитория развертывания из подраздела «Создание репозитория развертывания микросервисов» выше в этой главе, чтобы Argo CD мог получить доступ к вашим пакетам Helm.

Таблица 10.4. Значения для сервиса с информацией о рейсах

Раздел	Ключ	Значение
GENERAL	Application name	flight-info
GENERAL	Project	default
GENERAL	Sync policy	manual
SOURCE	Repository URL	YOUR_DEPLOYMENTS_REPOSITORY_URL
SOURCE	Path	ms-flights
DESTINATION	Cluster	in-cluster (https://kubernetes.default.svc)
DESTINATION	Namespace	microservices

Заполнив форму, нажмите кнопку Create (Создать).

Если у вас возникнут какие-либо проблемы, то обратитесь к документации Argo CD (https://oreil.ly/kZZJP) за инструкциями по настройке приложения.

После успешного создания приложения Argo CD отобразит приложение flight-info на панели мониторинга, как показано на рис. 10.14.

Рис. 10.14. Создано приложение с информацией о рейсах

Однако, хотя приложение и создано, оно пока не синхронизировано с объявлением `Deployment`, и приложение flight-info в нашем кластере не соответствует описанию в нашем пакете, потому что Argo CD пока еще не выполнил развертывание. Чтобы это произошло, щелкните кнопкой мыши на только что созданном приложении flight-info, нажмите кнопку Sync (Синхронизировать), а затем кнопку Synchronize (Синхронизация) в открывшемся окне, как показано на рис. 10.15.

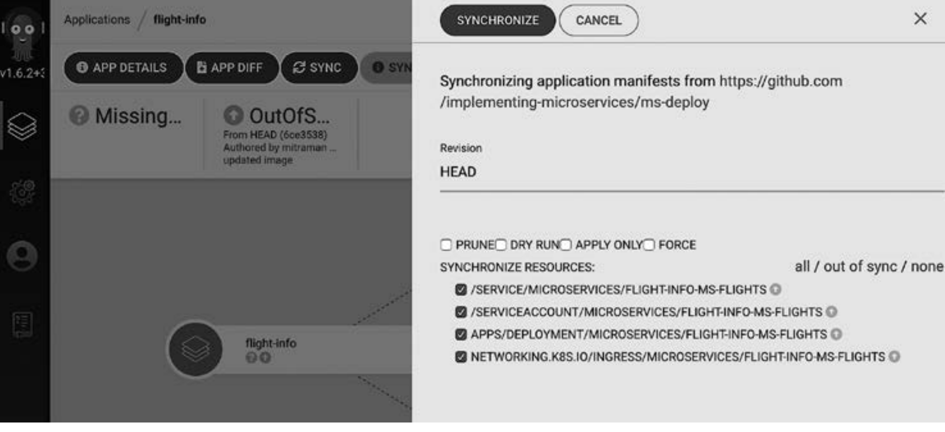


Рис. 10.15. Синхронизация приложения с информацией о рейсах

Когда вы нажмете кнопку Synchronize (Синхронизация), Argo CD выполнит все необходимое, чтобы привести приложение в состояние, описанное в пакете Helm. В случае успеха вы получите исправный, синхронизированный и развернутый микросервис, как показано на рис. 10.16.

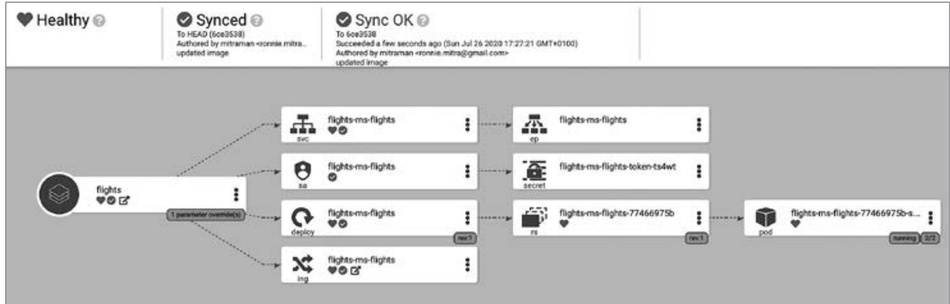
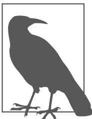


Рис. 10.16. Развернутый сервис с информацией о рейсах



Если вы увидите статус развертывания, отличный от Healthy (работоспособен), то щелкните на модуле Pod (крайний правый узел в дереве), чтобы получить список зарегистрированных событий и сообщений, которые могут помочь устранить проблему.

Наш контейнер был развернут в кластере Kubernetes, прошел проверки работо- и жизнеспособности и готов к приему запросов. Это серьезное достижение!

Теперь протестируем сервис с информацией о рейсах с помощью простого запроса.

Тестирование сервиса с информацией о рейсах

Теперь наш микросервис управления информацией о рейсах развернут и запущен в среде обкатки, размещенной в AWS. Чтобы проверить, как сервис обрабатывает запросы, нам нужно получить доступ к балансировщику нагрузки Traefik, который направит наш запрос в контейнерный сервис. Первое, что нам понадобится, — сетевой адрес балансировщика нагрузки. Поскольку мы не настраивали запись DNS, AWS автоматически предоставит случайный адрес. Чтобы узнать его, выполните следующую команду `kubectl`:

```
$ kubectl get svc ms-traefik-ingress
```

Вы должны получить что-то похожее на это:

```
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP
ms-traefik-ingress   LoadBalancer   172.20.149.191   ab.elb.amazonaws.com
```

Элемент `EXTERNAL-IP` — это адрес балансировщика нагрузки Traefik. Запишите его — он понадобится для создания тестового запроса.

Мы будем использовать curl для отправки запроса микросервису с информацией о рейсах. Если у вас нет локальной копии curl, то можете получить ее на сайте https://oreil.ly/xtfjJ. Для тех, кто не использовал эту программу раньше, отметим, что curl — это эффективный инструмент командной строки для отправки сообщений по заданному URL. Мы используем его, потому что он предлагает множество полезных функций, включая возможность задать заголовок `Host` в HTTP-запросе. Для нас это особенно ценно, поскольку для входного маршрутизатора нам нужно установить заголовок `Host` со значением **`flightsvc.com`**.

Выполните следующую команду, чтобы отправить сообщение с тестовым запросом сервису с информацией о рейсах (заменив *`{TRAEFIK-EXTERNAL-IP}`* адресом своего балансировщика нагрузки):

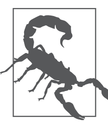
```
curl --header "Host: flightsvc.com" \
 {TRAEFIK-EXTERNAL-IP}/flights?flight_no=AA2532
```

В случае успеха вы получите подробную информацию об этом рейсе в формате JSON.

Вы можете использовать специальный инструмент тестирования API, такой как Postman или SoapUI, чтобы получить более удобную для пользователя форматированную версию ответного сообщения.

Только что созданный HTTP-запрос передается сервису входного шлюза, а тот пересылает его микросервису с информацией о рейсах согласно правилам маршрутизации, которые мы определили ранее в этой главе. Микросервис с информацией о рейсах извлекает сведения из подготовленного нами сервиса БД и возвращает результат через балансировщик нагрузки. С помощью этого запроса мы объединили все части нашей архитектуры и выполнили сквозное тестирование!

Все, что осталось, — выполнить очистку, чтобы не пришлось платить за неиспользуемые ресурсы AWS.



AWS выставляет счета за ресурсы EKS, даже если они не обрабатывают трафик, поэтому обязательно уничтожайте свою инфраструктуру, когда не используете ее.

Очистка

Как мы уже делали раньше, используем локальный клиент Terraform для уничтожения инфраструктуры. Перейдите в каталог, где находятся файлы Terraform среды обкатки, и выполните следующую команду:

```
infra-staging-env $ terraform destroy
```

После ее успешного завершения среда обкатки на основе Kubernetes будет уничтожена. Проверить уничтожение ресурсов можно с помощью интерфейса AWS CLI или веб-консоли AWS. Команды CLI, которые можно применить, мы показали в главе 6.

Резюме

В начале этой главы мы предупреждали, что нам предстоит проделать большую работу. И она принесла свои плоды — мы получили комплексное решение развертывания микросервисной архитектуры, которую создавали на протяжении всей книги. Нам также пришлось повторно использовать некоторые инструменты и методы, применявшиеся ранее, чтобы сделать больше за меньшее время.

В этой главе мы добавили в шаблон инфраструктуры поддержку зависимости от команд разработки микросервисов. Внедрили конвейеры сборки и интеграции в репозитории с кодом микросервисов, а также создали новый репозиторий и процесс развертывания сервисов, основанный на инструментах.

Надеемся, вы смогли увидеть, как решения, принятые ранее в отношении принципов, операционных моделей, инфраструктуры и проектирования, повлияли на решения, принятые на этапе развертывания. Все вместе они сформировали конечное состояние, которое позволило нам создать работающую реализацию.

Но настоящим испытанием для системы микросервисов станет проверка — как она справляется с обработкой изменений. Об этом мы расскажем в следующей главе.

ГЛАВА 11

Управление изменениями

К настоящему моменту мы создали систему микросервисов, оптимизированную для внесения изменений, и сделали это довольно быстро, воспользовавшись разнообразными инструментами, технологиями и репозиториями. В этой главе мы сделаем шаг назад и оценим созданную нами систему с точки зрения изменений. Посмотрим, как выглядят эти изменения в созданной нами системе. Перечислим типичные виды изменений, которые придется вносить, а также шаблоны и методы, помогающие в этом.

Изменения — важный фактор из-за оказываемого ими влияния на систему. Плохо разработанное ПО может принести организациям большие неприятности. Как мы подчеркивали в главе 1, одно из преимуществ системы микросервисов — возможность вносить изменения быстрее и безопаснее.

Кроме того, изменения всегда будут иметь свою стоимость. В программной системе она определяется как комбинация времени, денег и влияния на людей. Чтобы получить максимальную отдачу от своей системы микросервисов, нам необходимо минимизировать затраты на изменения и вносить лишь те, которые оказывают наибольшее влияние. Снижение затрат на изменения дает всем нашим командам больше свободы для экспериментов, оптимизации и совершенствования. Сосредоточение усилий на изменениях дает лучшие результаты при ограниченном бюджете.

Начнем со знакомства с основными видами предполагаемых изменений в системе микросервисов и лучшим способом принятия решений об изменениях.

Изменения в системе микросервисов

В системе микросервисов под изменением подразумевается расширение и улучшение функциональных возможностей, а не исправление проблем или ошибок. Это означает, что вы должны иметь возможность изменять систему, чтобы сделать ее лучше и получить больше пользы от созданного вами программного

обеспечения. Размышляя об изменениях в ПО, люди часто считают, что необходимость в них обусловлена внешними факторами — требованиями бизнеса или пользователей. Например, вот несколько распространенных причин для внесения изменений в систему:

- поддержка запуска нового продукта;
- устранение логической ошибки, ухудшающей пользовательский опыт;
- интеграция с новым партнером.

Все это важные причины для изменений, и наша архитектура должна способствовать такого рода изменениям, чтобы сделать их внесение максимально эффективным с точки зрения затрат. Но важно понимать, что архитектурный стиль микросервисов — это метод оптимизации. А это значит, что мы также должны учитывать внутренние факторы. Следующие изменения вытекают из нашего наблюдения за самой системой:

- разделение микросервиса для снижения сложности кода;
- автоматизация развертывания инфраструктуры, чтобы избежать дрейфа ее кода;
- оптимизация конвейера CI/CD для более быстрой доставки изменений.

Нет никаких сомнений в том, что вам придется поддерживать внешние изменения. Но, чтобы получить максимальную отдачу от системы, вам также необходимо научиться планировать и выполнять внутренние изменения. Хороший способ принять эту установку на постоянное совершенствование — организовать сбор данных и метрик и использовать их для принятия решений.

Ориентация на данные

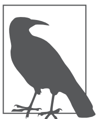
Классическая проблема при разработке программного обеспечения — чрезмерное увлечение инженерными решениями и преждевременная оптимизация. Так бывает, когда при разработке ПО или архитектуры мы пытаемся предусмотреть решение проблемы, которая практически никогда не возникает, или когда решение прогнозируемой проблемы обходится дороже, чем ущерб от ее проявления.

Все это может представлять опасность для системы микросервисов. Вот почему рекомендуется использовать данные и метрики для принятия решений о необходимости изменений, особенно тех, что направлены на внутреннее улучшение. Без данных вы будете строить догадки и в итоге усердно работать над улучшением тех частей системы, которые на самом деле не нуждаются в какой-либо доработке. Между тем другие насущные проблемы могут остаться незамеченными. С ограниченными ресурсами вы не можете позволить себе работать таким образом.

При использовании данных команды разработчиков могут принимать более обоснованные решения об изменениях, которые они хотят внести. Предприятия используют методологию целей и ключевых результатов (objective and key results, OKR), ключевые показатели эффективности (key performance indicators, KPI), оценки основных преимуществ, опросы удовлетворенности и показатели выручки, чтобы помочь себе принять стратегические решения и внести изменения.

Вам понадобится нечто подобное, чтобы обосновать свои планы по улучшению и оптимизации. Например, чтобы лучше понять перспективы улучшения, рассмотрите возможность сбора таких показателей проектирования, разработки и времени выполнения, как:

- время изменения для каждого микросервиса;
- частота изменений для каждого микросервиса;
- количество микросервисов, измененных в ответ на один запрос на изменение;
- строки кода в микросервисе (как точки данных, а не ограничение!);
- задержка времени выполнения в расчете на каждый микросервис;
- зависимости между микросервисами.



Мы не уделяли внимания наблюдаемости и журналированию в своей архитектуре микросервисов, потому что ограничены объемом издания и хотели сосредоточиться на некоторых наиболее основополагающих элементах. Но есть и хорошая новость: у вас есть все возможности расширить систему, чтобы она могла предоставить вам некоторые из описанных показателей.

В совокупности эти виды аналитических показателей могут дать вам более целостную картину о сферах, которые можно улучшить. Проанализировав данные, вы сможете принять решение о том, куда направить свои усилия. Конечно, вам также нужно будет сбалансировать вносимые улучшения с воздействиями, которые они окажут.

Влияние изменений

Изменение программного обеспечения может иметь много потенциальных последствий, но четыре из них вызывают наибольшие проблемы в современных организациях: время реализации, время на координацию, время простоя и влияние на потребителя. При оценке затрат на изменение в системе микросервисов рекомендуется учитывать эти основные области. Кратко опишем каждую из них.

- *Время реализации*. Основная часть затрат на любые изменения — это время, необходимое для фактического внесения изменений. Сюда входит время, необходимое для оценки текущего состояния, внесения желаемых изменений, их тестирования и обновления производственной среды. Важные факторы, влияющие на время реализации, — удобочитаемость, простота освоения и сопровождения изменяемых компонентов.
- *Время на координацию*. Реализация изменений почти всегда требует наличия какой-то формы коммуникации между командами. Время на координацию — подмножество времени реализации, но его стоит выделить отдельно. На самом деле это крайне важный аспект, о котором следует помнить всегда. Время на координацию может включать количество времени, затраченного на получение доступа к ресурсам, разрешения и согласия на внесение изменений, а также общие «организационные трения», возникающие при работе в большой организации. Время координации часто выступает фактором организационного устройства и структуры.
- *Время простоя*. Этот показатель определяет, как долго система или системный компонент остаются недоступными во время реализации изменений. Много лет назад простои были общепринятой частью процесса изменения программного обеспечения. Но времена и ожидания потребителей изменились. Сейчас на технологические команды оказывается все больше давления с целью минимизировать время простоя, необходимого для внесения изменений. Фактически в системе микросервисов принято стремиться к модели изменений с «нулевым временем простоя», согласно которой система остается постоянно доступной.
- *Влияние на потребителя*. Стоимость изменений влияет на пользователей системы. Об этом факторе часто забывают. Время простоя отражает одну из форм воздействия на потребителя, но даже модели с «нулевым временем простоя» могут иметь дорогостоящие последствия, которых можно было бы избежать. Например, изменение инфраструктуры может оказать широкомасштабное влияние на команды разработчиков микросервисов. Аналогично изменение интерфейса может привести к нарушениям в работе всех компонентов, которые его используют.

Архитектура программного обеспечения играет важную роль в формировании затрат и последствий изменений во всех четырех описанных областях. Иная грань процесса — это способ применения изменений. Архитектуры микросервисов, облачные инфраструктуры и методы DevOps позволили сделать огромный шаг вперед. Посмотрим на две современные модели развертывания, а также на более старую, сохранившуюся до наших времен.

Три шаблона развертывания

Есть множество различных способов применения изменений и развертывания программных компонентов. Прежде чем мы углубимся в возможность изменения созданной архитектуры, стоит рассмотреть три шаблона развертывания, которые мы будем использовать при внесении изменений в свою систему: сине-зеленый, канареечный и множественное исполнение. Мы начнем с рассмотрения сине-зеленых развертываний.

Сине-зеленое развертывание

В сине-зеленом (https://oreil.ly/zj7g-) развертывании поддерживаются две параллельные среды. Одна из них работает и принимает трафик, в то время как другая простаивает. Изменение применяется к среде, находящейся в режиме ожидания, и когда она будет готова, трафик перенаправляется в нее. Две среды меняются ролями: ожидающая становится активной, а активная — ожидающей, готовой к следующему изменению.

Этот шаблон развертывания позволяет безопасно вносить изменения в производственную среду, а перед переключением трафика не требуется беспокоиться о применении изменений к активной системе. Фактические цветовые обозначения сред не имеют значения — суть этого шаблона заключается в том, что две среды меняются ролями, становясь то активными, то бездействующими.

Преимущество этой модели заключается в ее способности значительно сократить время простоя, вплоть до уровня модели с «нулевым временем простоя». Однако поддержание двух сред требует осторожного обращения с системами хранения, такими как базы данных. Хранимые данные в таких системах необходимо синхронизировать, реплицировать или поддерживать полностью вне сине-зеленой модели.

Канареечное развертывание

Канареечное развертывание (https://oreil.ly/QXtSZ) аналогично сине-зеленому, но вместо поддержки двух полноценных сред предполагает одновременную работу двух версий компонента. В роли «канарейки» в этом шаблоне выступает измененная версия, она подобна «канарейке в угольной шахте» (https://oreil.ly/hr1Vk), которая предупреждает об опасности заранее. Например, чтобы выполнить канареечное развертывание веб-приложения, вы должны запустить новую канареечную версию веб-приложения параллельно с исходным веб-приложением.

Как и в сине-зеленом шаблоне, для поддержки канареечного развертывания требуется логика управления трафиком и маршрутизацией. После развертывания новой версии приложения часть трафика перенаправляется в новую версию. Трафик, попадающий в канареечную версию, может ограничиваться некоторым процентом от общего объема либо определяться уникальным заголовком или специальном идентификатором. В любом случае со временем все больше трафика направляется в канареечную версию, пока она в итоге не перейдет в полноценное рабочее состояние.

Канареечный шаблон похож на сине-зеленый, но у него есть дополнительное преимущество: он более «мелкомодульный». Вместо сопровождения полной копии среды можно сосредоточиться на меньшем, ограниченном изменении и внести его в работающую систему. Это может вызвать проблемы, если канареечная версия, развертываемая нами, повлияет на другие части нашей системы. Например, если канареечное развертывание изменяет общий системный ресурс по-новому, то обработка даже 1 % трафика в канареечной версии может иметь катастрофические последствия.

Но в системе, спроектированной с учетом возможности независимого развертывания компонентов, канареечный шаблон может работать довольно хорошо. Когда изменения вносятся в компоненты с четкими границами и владеющие собственными ресурсами, радиус возможных повреждений ограничен. Так что это хороший шаблон для тех, кто работает с подходящим типом архитектуры. Как мы увидим далее, канареечный шаблон хорошо подходит для архитектуры микросервисов, которую мы построили в этой книге.

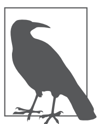
Многоверсионное развертывание

Последний шаблон рассматривает пользователей и клиентов как часть процесса изменений и заключается в параллельном выполнении нескольких версий. Рассмотренные нами шаблоны сине-зеленого и канареечного развертывания уже используют механизм временного запуска параллельных экземпляров (иногда называемый шаблоном *расширения и сжатия*). Но в обоих этих случаях вы обычно запускаете новые и старые экземпляры в частном порядке, не делясь подробностями о новой функции, пока не убедитесь в ее безопасности. Решение о маршрутизации — *неявное* и скрыто от пользователей системы.

Шаблон развертывания множества версий делает изменения более прозрачными для пользователей и клиентов системы. В этом шаблоне мы *явно* устанавливаем версию компонента или интерфейса и позволяем клиентам выбирать, какую версию компонента использовать. При таком подходе можно поддерживать одновременное использование нескольких версий.

Основная причина использования этого метода — возможность внесения изменений, требующих изменения зависимой системы. Данный шаблон применяется, когда известно, что люди, с которыми мы не координируем свои действия, тоже должны предпринять какие-то действия, чтобы завершить изменение. Классический пример такой ситуации — когда вы хотите внести изменения в API, которые нарушат работу клиентского кода. В этом сценарии управление миграцией всех сторон потребует значительных усилий по координации. Вместо этого мы можем организовать поддержку более старых версий, чтобы не ждать, пока каждый клиент изменится.

Использование этого подхода сопряжено с некоторыми серьезными проблемами. Каждая версия компонента, который мы внедряем, увеличивает затраты на сопровождение и усложняет нашу систему. Версии должны иметь возможность безопасно работать вместе, а параллельные версии — постоянно сопровождаться, поддерживаться, документироваться и оставаться безопасными. Эти накладные расходы могут стать головной болью для эксплуатации и замедлить внесение изменений с течением времени. В конце концов, вам придется заставить пользователей старых версий совершить миграцию и избавиться от устаревших версий.



Есть системы, которые почти никогда не прекращают поддержку старых версий. Например, на момент написания этой книги Salesforce SaaS API имел номер версии 49 и параллельно поддерживал 19 предыдущих версий!

Теперь у нас есть достойная основа для оценки воздействия изменений и набор типичных шаблонов развертывания, которые можно использовать для описания внедрения изменений. Теперь погрузимся в оценку созданной нами архитектуры с точки зрения изменений в инфраструктуре, микросервисах и данных. Для начала изучим возможность изменения платформы инфраструктуры.

Обзор нашей архитектуры

Следуя инструкциям в этой книге, вы сможете создать довольно продвинутую архитектуру микросервисов за достаточно короткий срок. Такая скорость разработки — свидетельство применения потрясающих инструментов, сервисов и программного обеспечения. Но быстрое создание чего-то бесполезно, если оно не выполняет ту работу, которую должно. Для нас это означает, что созданная архитектура должна неизменно сохранять работоспособность при внесении изменений.

В этом разделе мы проведем экскурсию по системе и подробно рассмотрим, как принятые нами решения повлияли на возможность изменения архитектуры. Обсудим изменения с точки зрения таких факторов, как затраты на реализацию, время на координацию, время простоя и влияние на потребителя, которые мы представили ранее в текущей главе. Для того чтобы упростить задачу, разделим архитектуру на три подсистемы: инфраструктуру, микросервисы и данные и рассмотрим каждую по очереди. Начнем с инфраструктуры.

Изменения в инфраструктуре

В главе 7 мы разработали платформу на основе Terraform для микросервисов, которая включает в себя сеть, Kubernetes и средство развертывания GitOps. Позже мы добавили базы данных MySQL и Redis, основываясь на потребностях команд микросервисов. Вполне реально ожидать, что инфраструктура платформы будет продолжать меняться по мере развития потребностей пользователей и команд, а также в результате изменения структуры спроса и бизнес-целей.

Изменения в нашей инфраструктуре мы можем разделить на две категории: добавляющие в платформу новые ресурсы и изменяющие существующие ресурсы. Создание новых ресурсов — форма расширения, которая мало влияет на работающую систему, в то время как изменениями существующих ресурсов необходимо тщательно управлять. Примерами добавления новых ресурсов в нашу архитектуру могут служить:

- внедрение новой инфраструктуры потоковой передачи событий с использованием AWS SNS для новых разрабатываемых микросервисов;
- подготовка экземпляра Elastic Container Service (ECS) и VPC для установки стороннего приложения;
- добавление новой операционной учетной записи в систему IAM.

Ниже приведены несколько примеров изменений существующих ресурсов:

- изменение архитектуры сети VPC, в которой развернут сервис EKS;
- обновление версии MySQL в нашем экземпляре RDS;
- изменение конфигурации кластера Kubernetes.

Мы должны учесть оба типа изменений, оценивая возможность изменения инфраструктуры. Начнем с рассмотрения затрат на реализацию.

Стоимость реализации изменений инфраструктуры

Стоимость реализации изменений инфраструктуры зависит от того, насколько сложно понять и выполнить изменение. Именно здесь помогает вклад, сделанный нами в проектирование инфраструктуры. Решения об использовании принципа неизменяемой инфраструктуры, постройке конвейера CI/CD и написании IaC в совокупности значительно снижают затраты на внесение изменений.

Когда придет время вносить изменения в инфраструктуру, вы можете использовать процесс, который благодаря реализованным ранее инструментам выглядит примерно так.

1. Решите, какие изменения в инфраструктуре хотите внести.
2. Определите, какой код инфраструктуры необходимо изменить (например, нужно ли создать новый модуль Terraform? Или просто обновить определение среды?).
3. Протестируйте изменение в среде разработки инфраструктуры.
4. Попробуйте развернуть приложения и микросервисы в обновленной инфраструктуре.
5. Запустите тесты и выполните выпуск (например, интеграционные тесты, тесты производительности и сквозные тесты).

Благодаря внедрению принципа IaC значительно улучшилась возможность изменения проекта инфраструктуры. Изменения всегда вносятся только через конвейер инфраструктуры, поэтому мы знаем, что, если этого нет в коде, значит, этого нет в инфраструктуре.

В каждой среде применяются одни и те же модули кода, поэтому мы знаем, что если изменения в инфраструктуре работают в среде разработки, то они должны работать и в производственной среде. Наконец, автоматизированный конвейер гарантирует, что инфраструктурный код и тесты будут выполняться одинаково последовательно и неоднократно.

В нашей системе мы исключили вариации из процесса изменений. Теперь у нас меньше поводов для беспокойства и мы можем сосредоточиться на самом изменении. Написание IaC требует немного больше первоначальных усилий, но отдача, когда дело доходит до изменений, делает это сто́ящим вложением.

В целом затраты на реализацию инфраструктуры с нашим подходом должны быть ниже, чем если бы мы просто вносили изменения непосредственно с помощью консоли AWS.

Стоимость координации изменений в инфраструктуре

При разработке операционной модели в главе 2 мы приняли важное решение о том, как будут выполняться инфраструктурные работы: единая команда под названием «команда разработки платформы» будет отвечать за проектирование, сопровождение и эксплуатацию облачной инфраструктуры. Централизация проектирования инфраструктуры в рамках этой команды означает, что мы будем нести относительно низкие затраты на координацию принятия решений, потому что нам не нужно будет добиваться консенсуса между всеми сторонами всякий раз, когда потребуется изменить инфраструктуру. Команда разработки платформы обладает достаточными полномочиями и автономией (и ответственностью!) в отношении изменений инфраструктуры.

На практике трудно предложить инфраструктурную платформу в чистом виде *«все как сервис»* (XaaS). Чтобы команды микросервисов могли использовать платформу для доставки изменений, обязательно потребуются поддержка, взаимодействие и согласие. Однако централизованный характер команды разработки платформы таит в себе потенциальную проблему. Как быть, когда команды потребуют внести противоречивые изменения? Как новые изменения тестируются во всех командах?

Модель платформы работает только в том случае, если существуют инструменты и процессы, позволяющие должным образом обеспечить режим взаимодействия с самообслуживанием и низкой координацией. Это требует большой предварительной и постоянно продолжающейся работы, и ее не следует недооценивать. Например, среда на основе Terraform не должна предлагаться командам микросервисов без соответствующей документации, отслеживания проблем и разумного уровня поддержки.

В полноценной системе микросервисов для любой крупной организации изменения инфраструктуры почти всегда влекут за собой дополнительные затраты на координацию из-за многоэтапности процессов. Потенциальное влияние плохого решения на инфраструктуру велико, поэтому обычно требуется проверка безопасности, выгодности и рисков, прежде чем можно будет начать развертывание изменений инфраструктуры. Один из практических способов снизить затраты на координацию — относиться к этим группам как к потребителям платформы и соответствующим образом разрабатывать решения.

Время простоя при внесении изменений в инфраструктуру

Трудно вносить изменения в инфраструктуру, не вызывая хотя бы минимального простоя. Это связано с тем, что инфраструктура — основополагающая часть программной системы. Например, как обновить сервер Kubernetes или внести серьезные изменения в сеть, не прибегая к временному отключению системы?

Созданная нами инфраструктура довольно легко поддается расширениям и дополнениям, достаточно лишь добавить код Terraform, который будет выполняться в конвейере. Однако изменить существующие части в нашей системе трудно, не вызвав хотя бы небольшого простоя.

Большой проблемой для нас является неизменный характер инфраструктуры. Чтобы внести даже небольшое изменение в компонент, нам нужно сначала удалить его. Это может быть сложно, если при этом мы надеемся продолжить обслуживать рабочие нагрузки и трафик.

Чтобы внести подобные изменения в инфраструктуру на месте, мы могли бы использовать сине-зеленый шаблон развертывания (см. пункт «Сине-зеленое развертывание» выше в этой главе). На самом деле мы бы даже сделали еще один шаг вперед и использовали шаблон развертывания Phoenix (https://oreil.ly/enM_P). Он похож на шаблон сине-зеленого развертывания, но вместо постоянной поддержки простаивающей среды он предполагает создание новых сред по мере необходимости с применением конвейера IaC.

Это означает, что мы могли бы создать новую среду со своими изменениями, развернуть в ней микросервисы после тестирования и, если все в порядке, переключить трафик на новую среду. Шлюз API или балансировщик нагрузки могли бы предоставить функции маршрутизации трафика, необходимые для такого маневра.

Но есть большая проблема — данные. У нас нет четкого разделения между экземпляром данных и экземплярами приложений. Для простоты мы разместили все свои базы данных в той же сети, что и экземпляры микросервисов. Это означает, что у нас нет простого способа развернуть новую среду, не выполнив ресурсозатратную работу по репликации данных. Это сильно усложнит процесс внесения изменений.

Если для вас важен принцип нулевого времени простоя, то вам необходимо пересмотреть проект инфраструктуры с точки зрения данных.

Влияние изменений в инфраструктуре на потребителя

Потребители наших приложений не будут напрямую взаимодействовать с инфраструктурой. Однако поскольку было принято решение предлагать инфраструктуру «как сервис» в рамках операционной модели, то нам необходимо учитывать влияние изменений на команды микросервисов.

Когда вы меняете какую-либо часть инфраструктуры, необходимо учитывать, как это изменение повлияет на все команды разработки микросервисов, потребляющие и использующие платформу как сервис. С увеличением количе-

ства микросервисов в системе это может потребовать значительных затрат на координацию.

Честно говоря, построенная нами архитектура почти ничего не предусматривает для решения этой проблемы. Если вы используете архитектуру как есть, то придется проделать определенную работу, чтобы убедиться, что изменения инфраструктуры не приведут к нарушению в работе существующих микросервисов. Всякий раз, внося изменения, вы должны будете проводить тестирование.

Чтобы снизить затраты на координацию, командам разработки платформы и микросервисов необходимо предусмотреть способ информирования об изменениях, поддержания автоматических тестов в актуальном состоянии и разделения ответственности за общую надежность и качество системы. Как всегда, для этого требуется сочетание топологии команд, архитектуры, а также хороших инструментов и технологий.

Единственное, чего мы можем ожидать наверняка, так это того, что количество микросервисов выйдет за пределы двух сервисов, которые мы использовали в нашем примере системы бронирования. Поэтому командам разработки микросервисов предстоит внести множество изменений. Рассмотрим их более подробно.

Изменения микросервисов

Большинство изменений, которые вам понадобится внести в систему, будут касаться самих микросервисов. Когда потребуется предложить новые продукты, изменить способ работы с пользователем или просто подстроить систему, скорее всего, вы будете вносить изменения в подсистему микросервисов. Это может означать создание нового микросервиса, обновление логики существующего сервиса или даже удаление, разделение или объединение сервисов.

Учитывая нашу действующую архитектуру, легко представить, что мы, возможно, захотим добавить больше функций в систему путешествий. Например, может появиться желание добавить в систему возможность бронирования железнодорожных билетов. В этом случае легко представить, что мы создадим новый кластер микросервисов и обновим API шлюза для поддержки этих новых возможностей.

Как мы видели во всех наших примерах, добавление чего-то нового — обычно самый простой вид изменений, которые можно внести. Однако ситуация усложняется, когда нужно изменить уже действующий сервис. В результате таких изменений могут возникнуть определенные затруднения:

- разделение микросервиса с информацией о рейсах на сервисы для внутренних и международных рейсов;
- добавление в сервис бронирования авиабилетов нового состояния «предварительного» бронирования;
- объединение микросервисов с информацией о рейсах и бронирования авиабилетов.

Во всех этих случаях управлять изменениями сложнее, потому что эти сервисы уже используются. К счастью, архитектура, которую мы создали вместе, отлично справляется с минимизацией этих воздействий. Взглянем на изменения в микросервисах через призму наших четырех ключевых последствий изменений.

Затраты на реализацию микросервисов

Основные затраты на реализацию изменения микросервиса связаны с возможностью понимания, сопровождения и тестирования кода. В нашей архитектуре мы приняли несколько важных решений, направленных на снижение затрат на реализацию.

- *Применять Event Storming для изменения размера микросервисов.* Метод Event Storming помог определить границы наших микросервисов, внутренне согласованные и учитывающие конкретные части предметной области. В результате код можно сделать проще, а изменения вносить меньшими порциями и чаще.
- *Все микросервисы используют* `microservice-bootstrap`. Этот фреймворк предоставляет командам последовательный способ документирования и тестирования разрабатываемых ими микросервисов. Сделав этот фреймворк обязательным, мы частично сократили нагрузку по внесению и тестированию изменений во всех организациях. Разработчики могут быстро ознакомиться с этим инструментом, а работа по тестированию и созданию сервисов может стать общей компетенцией для всех команд.
- *Применять конвейер CI/CD для микросервисов.* Это означает, что все наши изменения в коде тестируются, анализируются и проверяются. Конечным результатом является высокая вероятность получить код, пригодный для использования и сопровождения, к тому моменту, когда придет время вносить новые изменения.

В целом правильный размер сервиса и внедренный нами инструментарий DevOps должны значительно снизить затраты на внесение изменений в код микросервисов в нашей архитектуре.

Затраты на координацию микросервисов

Затраты на координацию могут стать большой проблемой при внесении изменений в программное обеспечение. Со временем простой фрагмент кода приложения может вырасти и содержать множество зависимостей от других библиотек, компонентов и систем. Эти зависимости затрудняют быстрое внесение изменений из-за организационных трений, обусловленных необходимостью работать со многими другими людьми и командами, чтобы понять, можно ли безопасно внести изменения.

В нашей архитектуре мы приняли несколько решений, которые должны помочь снизить эти затраты. В разделе «Что такое микросервисы» главы 1 мы дали определение микросервиса, в котором сделан акцент на независимости работ по проектированию и выпуску микросервисов. Такое мышление привело нас к принятию решений, которые могут повысить независимость команд микросервисов:

- каждым микросервисом владеет только одна команда;
- у каждого микросервиса собственный репозиторий и конвейер CI/CD.

Все вместе эти решения повышают автономность команд, которые вносят изменения в код микросервисов.

В дополнение к сокращению координации между командами наше решение «определять правильные» границы сервисов и ограничить размеры команды гарантирует, что затраты на координацию внутри команды также останутся относительно низкими. Справедливо будет сказать, что снижение затрат на координацию изменений кода микросервисов стало основной движущей силой созданной нами архитектуры.

Но есть две области, в которых трудно избежать затрат на координацию в нашей действующей архитектуре: события жизненного цикла и изменения интерфейса.

В главе 2 мы представили команду системы, которая несет ответственность за работоспособность и значимость системы в целом. Изменения, вносимые этой командой, могут привести к увеличению затрат на координацию. Например, что произойдет, когда команда системы решит, что два микросервиса должны быть объединены или, хуже того, что эти микросервисы должны принадлежать двум разным командам? В нашей архитектуре такого рода изменения потребуют гораздо большего согласования, планирования и взаимодействия, чем изменение кода отдельного микросервиса.

Мы сочли это приемлемым компромиссом по затратам. По нашему опыту, изменения, связанные с жизненным циклом и обслуживанием системы, относительно

редки по сравнению с изменением кода для отражения новых бизнес- или технологических требований. Имеет смысл оптимизировать модель изменений для тех типов изменений, которые, как мы ожидаем, будут происходить чаще.

Изменить код микросервиса — это одно дело, но часто приходится менять и его интерфейс. В этих случаях могут потребоваться дополнительные усилия по координации из-за контрактного характера API между потребителем и поставщиком. Мы более подробно рассмотрим этот фактор в пункте «Влияние микросервисов на потребителя» далее в этой главе.

Наконец, в главе 2 мы приняли решение создать единую команду по выпуску, которая будет отвечать за обновление производственной среды. Это решение имеет наибольшую вероятность оказаться ошибочным! Мы создали команду по выпуску, чтобы уделить особое внимание изменениям и затратам на координацию, которые часто сопровождают их. Кроме того, мы попытались снабдить команду разработчиков инструментами развертывания, чтобы свести к минимуму любое влияние на скорость их работы. Но в конечном счете если команда выпуска становится узким местом для изменений, то проект системы должен быть пересмотрен. Мы проведем переоценку топологии и инструментов, обеспечивающих цикл выпуска.

В целом затраты на координацию изменений микросервисов в нашей архитектуре невелики благодаря операционной модели, инструментам и проектным решениям, которые мы принимали на протяжении всей этой книги.

Время простоя микросервисов

Еще одна область изменений, которую мы оптимизировали, — минимизация времени простоя, необходимого при изменении отдельного микросервиса. Это связано с инструментарием и инфраструктурой, которые мы внедрили на уровне платформы. Ключом к снижению этих затрат выступает возможность применения канареечного шаблона развертывания (см. пункт «Канареечное развертывание» выше в этой главе) для выпусков микросервисов. Когда придет время выпускать новую версию микросервиса, вы сможете использовать установленные нами инструменты, чтобы выполнить процесс изменения, описанный ниже.

1. Развернуть новую версию микросервиса канареечным способом вместе с существующей версией.
2. Реализовать правило маршрутизации трафика, отправляющее небольшой процент трафика в новую версию.
3. Понаблюдать за работой новой версии и убедиться, что результаты соответствуют ожиданиям.

4. Активизировать канареечный микросервис, направив ему весь трафик.
5. Дождаться, когда старая версия обработает все запросы, и удалить ее.

Этот шаблон подходит для большинства изменений, которые понадобится внести, и вы сможете использовать Argo CD для оркестрации действий канареечного сервиса. Однако будьте внимательны при использовании этого шаблона. Новая версия микросервиса внесет изменения, которые могут повлиять на более старую. Например, если новая версия изменяет данные в общей базе данных, то убедитесь, что изменения совместимы с предыдущими запущенными версиями.

Влияние микросервисов на потребителя

До сих пор основное внимание мы уделяли изменениям в коде микросервисов. Логика, проверка и поведение сервиса отражены в коде, поэтому именно в него будет вноситься большая часть изменений. Но иногда требуется внести изменения в интерфейс (или API) микросервиса, и это может вызвать серьезные проблемы.

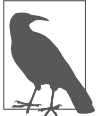
Изменение интерфейса микросервиса практически неизбежно. В конечном счете вам захочется изменить параметры операции или возвращаемые данные из вызова. Проблема в том, что, поскольку другие сервисы и компоненты начинают зависеть от интерфейса, даже небольшие изменения могут потребовать значительных усилий по переделке от всех участников.

На самом деле мы ничего не предусмотрели в нашей архитектуре, чтобы уменьшить влияние изменений на потребителя. Лучший способ уменьшить влияние изменений API на потребителя — придерживаться некоторых проверенных методов проектирования: не менять то, что уже выпущено, писать клиентский код, допускающий появление новых данных, и не делать новые входные параметры обязательными.

Наш любимый источник рекомендаций по проектированию API — Майк Амундсен. Если вы заинтересованы в создании эволюционирующих API, то мы рекомендуем изучить шаблоны изменения API в его книге Design and Build Great Web APIs (Pragmatic Bookshelf, 2020).

В дополнение к этим принципам проектирования некоторые специалисты по микросервисам практикуют контрактное тестирование, помогающее минимизировать затраты на координацию между командами при изменении интерфейсов. При контрактном тестировании потребители и поставщики совместно используют контракт, в котором описывается, как будет применяться

интерфейс. Это позволяет поставщикам независимо тестировать контракты и проверять влияние изменений в них на существующих клиентов API.



Чтобы запустить нашу систему как можно быстрее, мы не включили компонент контрактного тестирования в архитектуру. Но многие практикующие специалисты добились успеха, используя Pact (https://pact.io) для тестирования контрактов, ориентированных на потребителя. Такие инструменты, как Pact, позволяют потребителям и поставщикам постоянно обмениваться информацией и тестировать изменения, которые вносятся в их интерфейсы.

Но даже при использовании контрактного тестирования велика вероятность, что вам придется внести изменения, которые нарушат работу чьего-то кода. В этом случае вам нужно будет реализовать некую форму шаблона множественных версий (см. пункт «Многоверсионное развертывание» выше) и поддерживать старый микросервис до тех пор, пока команда клиентов не внесет необходимые изменения.

В целом наша архитектура мало что делает для снижения затрат на изменения, влияющие на потребителя. Изменение API — сложная задача, и потребуется хорошее проектное мышление и хорошее планирование, чтобы сделать эти изменения возможными. Еще одна опасная область — данные, и о ней мы поговорим далее.

Изменения данных

Один из самых сложных аспектов поддержки архитектуры микросервисов — работа с данными. Модели данных, как известно, трудно изменить. Уровень хранения — важная часть любой программной системы, но, когда приходит время изменять структуру данных, все может усложниться. Программные компоненты становятся зависимыми от используемых ими систем обработки данных, и их изменение может привести к большим затратам и влиянию на систему.

Мы попытались создать решения, улучшающие эту ситуацию и снижающие затраты на изменение модели данных. Взглянем на архитектуру данных, которую мы построили, через нашу призму четырех видов затрат на изменения.

Затраты на реализацию

На самом базовом уровне затраты на изменение модели данных зависят от сложности структуры, форматов и взаимосвязей, а также инструментов или языка, необходимых для внесения изменений. Сложность модели может увеличиваться, когда есть непростые значения, много различных типов данных и уникальных

ключей. В действительности затраты связаны с необходимостью понять саму модель данных, чтобы безопасно вносить изменения.

Мы почти ничего не предусмотрели в нашей архитектуре, чтобы предотвратить чрезмерное усложнение модели данных. Но решили, что микросервисы должны владеть своими данными. Само по себе это решение должно помочь ограничить область применения и размер модели и тем самым уменьшить затраты на изменение кода.

Таким образом, как и в случае с изменениями кода, отдав предпочтение независимости, вы должны получить значительную выгоду с точки зрения затрат на реализацию. Но, как и в случае с кодом, вам нужно будет продолжать измерять затраты на реализацию, чтобы гарантировать, что сервис и его модель данных не вырастут до размеров, сводящих на нет преимущества строгих границ.

Затраты на координацию

Еще большее преимущество приоритизации независимости — сокращение затрат на координацию. Решив, что микросервисы должны владеть своими данными, мы получили возможность свободно вносить изменения в структуры данных без согласования с другими командами или владельцами систем. Это резко контрастирует с более традиционными моделями, в которых несколько команд могут использовать общие данные, а изменения должны тщательно согласовываться со всеми пользователями данных.

Однако будьте осторожны: независимый подход к обработке данных сопряжен со скрытыми издержками. Мы оптимизировали нашу архитектуру для частых автономных локальных изменений. Это сделало общесистемные изменения более дорогостоящими. Например, если потребуется изменить глобальное определение идентификационного кода авиакомпании, то вам придется согласовать это изменение со всеми группами, которые внедрили модель данных, использующую этот код. В нашей архитектуре это может обойтись дороже, чем если бы мы просто использовали общую базу данных.

Хорошим источником, позволяющим понять распределенные шаблоны данных, является книга Designing Data-Intensive Applications[1] Мартина Клеппмана (O'Reilly, 2017) (https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/).

[1] *Клеппман М.* Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2018.

Мы решили оптимизировать систему для локальных изменений, так как, по нашему опыту, их частота выше. Но вам нужно будет изменить это решение, если система, которую вы создаете, может подвергнуться радикальным глобальным изменениям.

Если вы обнаружите, что часто вносите изменения сразу в несколько моделей данных, то это может быть признаком необходимости переоценки границ микросервисов.

Время простоя

Наша независимая модель данных дает существенные преимущества, когда дело доходит до координации. Но она не предполагает возможности изменения с нулевым временем простоя. Это особенно верно для базы данных MySQL, которую использует микросервис с информацией о рейсах.

Корень нашего ограничения заключается в использовании общего экземпляра базы данных для обслуживания нескольких реплик микросервиса. Когда придет время вносить изменения в модель данных, будет трудно сделать это, не повлияв на микросервисы, запущенные в данный момент. В меньшей степени это относится к хранилищу Redis, которое использует сервис бронирования. Но все равно стоит быть осторожными с внесением изменений, которые приведут к нарушению существующих версий.

В тех случаях, когда необходимо внести разрушительное изменение модели данных, самым простым вариантом может быть уничтожение существующих версий микросервисов и замена их новыми экземплярами, реализующими изменение данных. В среде Kubernetes это можно выполнить, оказав минимальное воздействие на сервис. Но если ни о каких простоях не может быть и речи, то потребуется более продуманное сине-зеленое развертывание.

Влияние на потребителя

Поскольку мы решили, что микросервисы должны владеть своими данными, влияние изменения модели данных ограничивается самим сервисом. Соответственно, мы можем свободно вносить изменения, не оказывая прямого влияния на потребителя сервиса. Поскольку модель данных инкапсулирована в микросервис, команды разработки микросервисов будут иметь больше автономии в части внесения изменений в свою модель, хотя, как мы подчеркивали выше, эти изменения могут потребовать небольшого времени простоя.

На практике изменение модели данных часто требует изменения кода и иногда даже интерфейса. Но мы вольны разделить или упорядочить эти изменения, чтобы сначала изменить модель данных, а потом реализовать изменения, непосредственно влияющие на потребителей.

Резюме

В целом созданная нами архитектура спроектирована так, чтобы упростить изменения и уменьшить затраты, связанные с ними. Изменения могут исходить из внешних или внутренних источников, но главное для нас — снизить затраты и последствия и тем самым дать командам больше свободы для улучшения системы, продуктов и впечатлений, которые обеспечивает система.

Мы рассмотрели нашу архитектуру инфраструктуры, кода, API и данных с точки зрения изменений. В этой главе мы увидели, что решения, принимавшиеся на протяжении всей книги, в совокупности расширили возможности изменения данной системы. Одни наши решения были компромиссами, принятыми для оптимизации определенных типов изменений. Другие — компромиссами, обусловленными ограничениями формата книги!

Независимо от причин теперь мы научились создавать архитектуры микросервисов и оценивать их полезность и пригодность. Единственное, что осталось сделать с нашей архитектурой, — улучшить ее, о чем мы и поговорим в следующей главе.

ГЛАВА 12

Конец путешествия (и новое начало)

Поздравляем, вы добрались до последней главы! Хотя наше совместное путешествие заканчивается, мы надеемся, что для вас это лишь начало долгого и плодотворного пути успешного внедрения микросервисов в реальных проектах. Мы не ищем оправданий тому факту, что являемся поклонниками архитектуры микросервисов и преимуществ, которые она может дать при развертывании в правильном контексте, с правильными намерениями и навыками. Это ни в коем случае не единственный вариант выбора, и его никогда не следует реализовывать, не понимая всех последствий, но он, безусловно, может стать очень эффективным средством в вашем арсенале архитектурных инструментов.

Мы были свидетелями многих успешных проектов микросервисов. Но и неудачных попыток внедрения микросервисов было предостаточно. Этой книгой мы хотели бы увеличить шансы читателя на успех, если он решит реализовать свою систему в стиле микросервисов. Мы постарались предоставить пошаговые практические рекомендации о том, когда, почему и как развертывать микросервисы, объяснить основные концепции и продемонстрировать реализацию этих концепций на простых примерах. Надеемся, что нам удалось достичь цели: превратить абстрактные концепции в более доступное пошаговое объяснение. Но самое главное, мы надеемся, что вам понравилась эта книга, даже притом, что в ней содержится лишь несколько ключевых идей, которые вы сможете использовать при реализации собственных систем.

Прежде чем мы расстанемся, хотелось бы поделиться заключительными мыслями, обобщающими наше понимание архитектурных решений в микросервисах, и подходом, который мы рекомендуем для постоянной оценки прогресса преобразования, если вы все-таки решите приступить к реализации.

О сложности и упрощении использования микросервисов

На протяжении всей книги мы утверждали, что микросервисы лучше подходят для реализации больших, сложных и постоянно меняющихся систем. Интуитивно это утверждение имеет смысл: архитектура микросервисов сама по себе непроста, поэтому стоит начать это путешествие — особенно если оно поможет решить что-то еще более сложное. Но какова природа сложности и как именно микросервисы уменьшают ее, если вообще уменьшают?

В своей основополагающей работе по сложности программного обеспечения — статье «Серебряной пули нет» (http://worrydream.com/refs/Brooks-NoSilverBullet.pdf), опубликованной в 1986 году, Фред Брукс метко отмечает:

> «Нет ни одной разработки ни в технологии, ни в методике управления, которая сама по себе обещала бы хотя бы на порядок повысить производительность, надежность и простоту».

Брукс продолжает развивать тему, объясняя, что причина этого явления — существенная сложность программных систем. В то время как в любой кодовой базе всегда есть некая «случайная сложность» (сложность, обусловленная выбором реализации), большая часть сложностей программных систем не случайна — она обусловлена сложностью моделирования самой предметной области как таковой. Это «существенная сложность», под которой Брукс подразумевает сложные наборы данных, взаимосвязи между элементами данных, алгоритмы и потоки вызовов, представляющие модель, которые пытается представить система. Если бы мы пытались упростить систему сверх ее существенной сложности, то отошли бы от ее основной модели, и это была бы уже другая система.

Когда речь идет о микросервисах, пионеров часто привлекает перспектива, позволяющая упростить создание сложных систем: потерпите и будет легче! Многие из нас хотели бы, чтобы микросервисы облегчили нам работу, а не усложняли жизнь, поэтому неудивительно, что обещание «будет легче» является эффективным мотиватором. Легко даются быстрые, импровизированные объяснения: внедряя «большую» систему в виде набора множества простых микросервисов, мы упрощаем весь процесс! Скептики могут сразу заметить, что, несмотря на простоту и небольшой размер каждого микросервиса, нельзя ожидать, что организация большого их количества в согласованную сложную систему будет легкой задачей. И они правы. Но что более важно, для тех из нас, кто читал «Серебряной пули нет» Брукса, возникает еще один неприятный вопрос: нарушают ли микросервисы гипотезу Брукса об отсутствии способов устранить существенную сложность? Или архитектура микросервисов направлена исключительно на случайную часть сложности

системы и выполняет такую удивительную работу, что мы все еще можем ощущать улучшение?

Правда в том, что ни то ни другое не верно. Концепция микросервисов не просто случайная сложность и не методология улучшения чистоты программирования, а принципиально другой подход. И нет, он не отменяет наблюдения Брукса. Скорее достигает своих целей в соответствии с этим наблюдением. Видите ли, вы не можете устранить существенную сложность, но можете перенести ее из одной части системы в другую. В этом не было бы ничего особенного, если бы только разные части системы не требовали разного объема прилагаемых усилий.

Проще говоря, усилия, прикладываемые к созданию любой программной системы, делятся на две части — ее реализацию (в виде кода) и управление ею (развертывание и оркестрация). Мы можем упростить код, разбив его на множество небольших микросервисов. Но такой шаг пропорционально усложнит управление системой. Казалось бы, мы почти ничего не добились, так как, упростив одну часть, усложнили другую. Но на самом деле такой сдвиг сложности может быть весьма полезным, если есть возможность автоматизировать «усложнившуюся» часть, но нет возможности автоматизировать упростившуюся часть. Возросшая сложность операционного управления имеет гораздо меньшее значение, если его можно автоматизировать. И это действительно достижение.

За последнее десятилетие или около того мы значительно продвинулись в автоматизации управления программными системами. Обширный арсенал инструментов автоматизации, таких как Ansible, Puppet, Chef, Terraform, Docker и Kubernetes, вместе с бессерверными функциями и широким спектром облачных сервисов, избавляющими нас от необходимости даже думать об этом, значительно упростили управление сложными системами, превзойдя все, что Брукс мог себе представить в 1986 году. Однако разработка и написание кода остались почти такими же сложными, как и в 1980-х годах. Не поймите нас неправильно: конечно, были некоторые достижения, но ничего существенного. Поэтому если мы перенесем сложность с этапа программирования на этап управления, то сможем упростить задачу нетривиальными способами.

Микросервисы могут обеспечить упрощение

Архитектура микросервисов может быть гораздо проще, чем ее альтернативы, при реализации сложных систем. Это не нарушает принципа Брукса «Серебряной пули нет», поскольку микросервисы не устраняют существенную сложность. Скорее этот архитектурный стиль направлен на перенос сложности из области проектирования и программирования, которую мы пока не можем автоматизировать, в область управления, в которой все очень хорошо с автоматизацией. Чистая прибыль может быть значительной.

Квадрант микросервисов

Углубимся в тему сложности. Теория систем различает сложные и усложненные системы. Эта идея была дополнительно расширена и популяризирована для принятия решений во фреймворке Cynefin (https://oreil.ly/3Wx_M). Сложная система может быть весьма замысловатой и трудной для понимания, но по своей сути предсказуемой и основанной на конечном количестве четко определенных правил. Напротив, усложненная система по своей сути не детерминирована, состоит из множества компонентов, которые взаимодействуют достаточно свободно и, следовательно, могут порождать непредсказуемое поведение. Если бы мы классифицировали монолиты и микросервисы в этих терминах, то монолиты считались бы сложными, тогда как микросервисы гораздо больше соответствуют определению усложненных систем.

Еще одна интересная классификация — противопоставление понятий «легкое» и «простое». Как с энтузиазмом подтвердят большинство проектировщиков, эти, казалось бы, синонимичные прилагательные не могут быть более разными в контексте проектирования. Простые вещи, как известно, сложно спроектировать (вспомните оригинальные iPod и iMac от Apple или такое простое изобретение, как компьютерная мышь), в то время как легкие конструкции не обязательно просты в использовании.

Объединив эти две точки зрения по оси архитектуры и реализации, пару лет назад мы создали «квадрант микросервисов», который вы можете видеть на рис. 12.1.

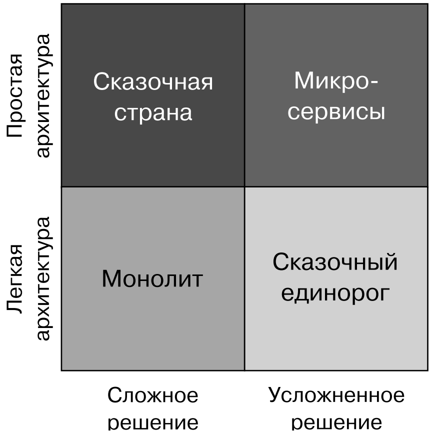


Рис. 12.1. Квадрант микросервисов (источник: https://oreil.ly/I05t8)

Эта шутливая диаграмма (любимая деловыми изданиями и выпускниками MBA) утверждает: думая об общем континууме сложности и простоты, мы можем распределить различные типы решений по четырем квадрантам:

- микросервсисы можно отнести к усложненным решениям, но к простым проектам (архитектурам);
- монолиты можно отнести к сложным решениям, но легким (не обязательно простым) архитектурным проектам.

В отношении двух других квадрантов можно сказать следующее.

- Многие разработчики программного обеспечения хотели бы иметь решение с простой архитектурой и в то же время предсказуемой, пусть и сложной реализацией. Вероятно, это была бы «сказочная страна» — немикросервисные реализации, которые являются элегантными и успешными, поэтому нельзя называть их монолитами во избежание устоявшейся негативной коннотации. Откровенно говоря, они довольно редки. Если ваша система постоянно и быстро меняется, то достижение решения в этом секторе равносильно достижению мечты.
- В нижнем правом квадранте мы имеем ситуацию, когда мы отделались легким проектом (например, с минимальными усилиями) и в итоге получили усложненную реализацию, которая каким-то образом все еще функционирует, несмотря на легкую архитектуру. Что ж, это был бы сказочный единорог во многих отношениях, включая потребность в десятках разработчиков для его поддержки и сопровождения. Но мы уверены, что такие вещи тоже существуют.

Квадрант микросервисов дает краткое представление о том, где находятся микросервисы и монолитные решения с точки зрения архитектурной простоты по сравнению со сложностью реализации.

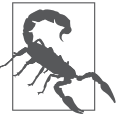
Обсудив природу архитектуры микросервисов через призму сложности, мы хотели бы показать читателю еще одну важную перспективу: как думать о трансформации микросервисов с течением времени.

В главе 11 мы обсудили роль архитектуры микросервисов, помогающую командам справляться с изменениями в сложных системах, и методы управления изменениями при внедрении микросервисов. Но есть еще один важный аспект, касающийся изменений в микросервисах: трансформация, которую организация в целом должна пройти при переходе от культуры, не связанной с микросервисами, к принятию этой новой организационно-технической структуры. В следующем разделе мы обсудим, как добиться успеха в трансформации на пути к микросервисам, взглянув на них целостно и избежав ловушки технологических шор.

Оценка прогресса трансформации на пути к микросервисам

Обсуждая переход на микросервисы, важно помнить, что мы говорим о стиле, который включает сложные технологии и крайне разрушительные культурные преобразования организации. Если не управлять им аккуратно, то вероятность ошибиться становится намного выше вероятности случайного везения. Если вы следили за различными критическими публикациями о микросервисах в течение последних нескольких лет, то могли заметить довольно четкую закономерность. Сначала компания приняла архитектуру микросервисов и написала веселый пост в блоге о ее преимуществах и перспективах, а через несколько лет за этой публикацией последовал пост с жалобами на сложность микросервисов и восхвалением возврата к монолиту. Для некоторых команд, проектов или компаний микросервисы действительно могут быть неправильным выбором, реальность такова, что фактическое несоответствие архитектуры не всегда становится причиной неудачи. Чаще всего причиной неутешительных результатов является плохая реализация.

Не существует готового программного обеспечения, способного волшебным образом превратить нашу систему в «микросервисы» к утру, которое команды могут просто купить у поставщика или установить в виде приложения с открытым исходным кодом. Более того, не существует даже какого-то строгого набора политик и руководств, гарантирующих успех. На самом деле многие желательные черты архитектур микросервисов (https://oreil.ly/_aFU8), такие как независимое развертывание, децентрализованное управление, автоматизация инфраструктуры и эволюционная архитектура, не поддаются прямому измерению. И никакая команда не в состоянии преуспеть в развитии их всех сразу или с легкостью оценить свой прогресс! Для их созревания требуется много времени и изрядное количество терпения, и они редко получаются идеальными. Не следует ставить цель совершенствования микросервисов в самом начале процесса преобразований.



Один из самых разрушительных шагов, которые организация может предпринять на ранних стадиях внедрения микросервисов, — создать «полицию микросервисов», которая будет строго контролировать соблюдение всех принципов и характеристик микросервисов. Переход на микросервисы — длительный процесс; это путешествие, требующее терпения и принятия взвешенных решений.

Подход, который необходимо принять командам при рассмотрении их уровня зрелости по отношению к характеристикам микросервисов, во многом аналогичен философии, которую мы описали в главе 4 при обсуждении правильного

размера микросервисов: размер и степень детализации микросервиса органично эволюционируют со временем, и попытка начать с целевой детализации на ранней стадии принесет вред. Точно так же весьма рискованно преждевременно настаивать на «идеальной» реализации микросервисов, поскольку это затрагивает такие характеристики, как возможность независимого развертывания и автоматизация на слишком ранней стадии процесса преобразования. Вместо этого важно, чтобы команды оставались прагматичными и задавали себе вопросы, перечисленные ниже.

- Kubernetes, несомненно, является ведущим решением для оркестрации контейнеров, но обладают ли наши разработчики знаниями и навыками для работы с ним? Даже если этот фреймворк поддерживается нашим решением для облачного хостинга? Или мы должны начать с чего-то гораздо более простого (например, AWS ECS)?
- Насколько автоматизированной должна быть наша инфраструктура на первых этапах? Какой уровень способности к самовосстановлению совершенно точно потребуется в начале?
- Управление какими системами можно делегировать поставщику облачных услуг (например, базы данных, потоковую передачу событий и т. д.), даже если потом мы вернем их себе? Обязательно ли начинать с совершенно новой системы баз данных или на первых порах можно использовать менее эффективную, но облачную БД, чтобы сократить затраты на сопровождение?

В большинстве случаев правильным решением в отношении этих и подобных вопросов будет дать себе слабину в первые дни. Возможно, разумнее придерживаться «скучных» технологий и избегать обновления MySQL до Cassandra или замены Java на Golang одновременно с внедрением микросервисов, особенно если ваши команды не знакомы с этими новыми технологиями. Вместо этого команды должны сосредоточиться на том, что имеет значение для бизнеса, и избегать погружения в бесконечные циклы настройки инфраструктуры, обновления технологического стека и экспериментов с новыми классными инструментами, которые серьезно задерживают создание значимой для бизнеса системы. Такие задержки могут привести к тому, что заинтересованные стороны свернут работу по преобразованию еще до того, как она будет должным образом начата.

Чрезвычайно важно помнить, что архитектура микросервисов — это путешествие, а не просто пункт назначения. В данном путешествии важнее траектория прогресса, и, как бы удивительно это ни звучало, текущее состояние имеет гораздо меньшее значение. Это особенно верно в первые дни работы по преобразованию.

В главе 1 мы отметили, что минимизация затрат на координацию — основной метод архитектуры микросервисов. Этот показатель настолько фундаментальный, что команды, которые смогут продемонстрировать тенденцию к снижению потребности в координации, преуспеют независимо от того, скольких принципов Ньюмана, Льюиса, Фаулера или Митры/Надареишвили они придерживаются изначально. Пока команды движутся в правильном направлении, траектория будет вести к победе в долгосрочной перспективе. Этот подход аналогичен концепции функций пригодности, описанной в книге *Building Evolutionary Architectures*[1] Нила Форда, Ребекки Парсонс и Патрика Куа (O'Reilly) (https://www.oreilly.com/library/view/building-evolutionary-architectures/9781491986356).

Как узнать, находимся ли мы на правильной траектории? Конечно, полезно понимать, что затраты на координацию — главный враг, но мы не можем напрямую измерить «затраты на координацию» как значение. Некоторые команды пытаются измерить «скорость» или «безопасность», но это в равной степени проблематично, поскольку эти значения являются производными, а измерения — необоснованными. Вы почти наверняка заметите приличное увеличение скорости и безопасности, но, говоря о причинно-следственной связи, с чем вы собираетесь сравнить новую скорость? Никто не строит одну и ту же систему один раз как монолит, а затем как архитектуру микросервисов. Любое увеличение скорости интуитивно ощутимо, но неизмеримо. То же относится и к попыткам измерения повышения безопасности.

Вместо этого мы предлагаем измерять три значения, два из которых напрямую связаны с траекторией повышения автономии команды, а третье оценивает общую эффективность команд разработчиков программного обеспечения (как описано в книге Accelerate[2] Н. Форсгреном, Дж. Хамблом и Д. Кимом (IT Revolution Press) (https://www.oreilly.com/library/view/accelerate/9781457191435)):

- средний размер автономной команды среди всех команд;
- средняя продолжительность времени, в течение которого автономная команда может работать без остановки в ожидании другой команды (ожидание обычно вызвано критической зависимостью);
- частота успешных развертываний.

---

[1] *Форд Н., Парсонс Р., Куа П.* Эволюционная архитектура. Поддержка непрерывных изменений. — СПб.: Питер, 2019.

[2] *Хамбл Дж.* Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации.

При здоровой трансформации микросервисов по правильной траектории вы должны увидеть постепенное уменьшение размера автономных команд и увеличение времени, в течение которого команды могут работать независимо. Например, вы можете заметить, что средний размер автономной команды в вашей организации раньше составлял от 15 до 20 человек, а после внедрения микросервисов он начинает постепенно уменьшаться до 10, 8, 6...

Точно так же должно наблюдаться снижение частоты координационных тупиков. *Координационный тупик* — это остановка, во время которой автономная команда ожидает от другой предоставления общих ресурсов, например, когда команда инфраструктуры подготовит высокодоступный кластер Kafka или Cassandra или группа проверки безопасности завершит аудит кода. Другой распространенный пример остановки команды — ожидание результатов координационного совещания, на котором различные заинтересованные стороны принимают важное решение.

Планирование таких встреч может занять много времени из-за различных приоритетов заинтересованных сторон. Отслеживание количества зависимостей, которые команде необходимо прояснить перед выпуском кода в производство — еще одна величина, которую стоит измерить. Другой важный пример событий, которые необходимо отслеживать, — частота приостановки команд в ожидании, пока другие внесут в код изменения, вызванные изменением в общей модели данных. Причины и продолжительность остановок будут варьироваться в зависимости от организации и бизнес-контекста. Важно отслеживать как типы причин, так и продолжительность остановок, чтобы извлечь значимые практические уроки и внести улучшения.

Третий показатель, частота развертывания, не измеряет напрямую затраты на координацию, но является общей метрикой, которая, как было научно доказано Форсгреном и др. (https://www.oreilly.com/library/view/accelerate/9781457191435), служит убедительным показателем гибкости команды. По нашему опыту, применительно к независимо развертываемым микросервисам он также может указывать на правильность траектории преобразования микросервисов.

Последовательно измеряя три показателя и гарантируя продвижение трансформации по правильному пути, команды могут перестать беспокоиться о достижении совершенства в каждой отдельной характеристике микросервисов, высвобождая силы и время для развития долгосрочного успеха в работе.

Резюме

В этой заключительной главе мы поделились с вами своими мыслями о микросервисах. Микросервисы могут упростить сложные системы, но они не панацея, и важно понимать, что конечный эффект достигается за счет перемещения сложности, а не ее волшебного устранения. Такие утверждения также помогают прояснить, что мы подразумеваем под «усложненностью», чем отличаются «усложненные» системы от «сложных» и какую роль играют «легкие» и «простые» архитектурные подходы в классификации различных подходов к доставке систем.

Затем мы поделились своим мнением о важности терпения и долгосрочных перспектив во время перехода к микросервисам. Это путешествие и марафон, а не спринт, и команды, намеревающиеся добиться успеха, должны иметь надлежащие инструменты и гораздо больше концентрироваться на траектории трансформации, чем на текущем состоянии. Возьмите за правило измерять некоторые надежные показатели, помогающие убедиться, что вы все еще на правильном пути и ваша траектория в порядке.

Мы надеемся, что вам понравилась эта книга, что вы нашли в ней новые для себя практические рекомендации и получили удовольствие от знакомства с кодом и примерами.

Мы желаем вам успехов на вашем собственном пути преобразования микросервисов и хотели бы услышать от вас, чему вы научились, начав внедрять микросервисы в свою работу.

Всего наилучшего.

Об авторах

Ронни Митра — автор книг, стратег и консультант с более чем 25-летним опытом работы с веб-технологиями и технологиями связи. Соавтор книг *Microservice Architecture* и *Continuous API Management*[1] (обе изданы O'Reilly).

Иракли Надареишвили — вице-президент по основным инновациям в Capital One Financial Corporation, возглавляет команды, ответственные за создание современной облачной банковской платформы, основанной на микросервисах. Ранее был соучредителем и техническим директором медицинского стартапа ReferWell, а также занимал руководящие должности в CA Technologies и NPR. Соавтор книги *Microservice Architecture* (O'Reilly). Вы можете подписаться на Иракли в Твиттере: @inadarei.

[1] *Амундсен М., Меджуи М., Митра Р., Уайлд Э.* Непрерывное развитие API. Правильные решения в изменчивом технологическом ландшафте. — СПб.: Питер, 2023.

Иллюстрация на обложке

На обложке изображен сверкающий колибри *(Colibri coruscans)*. Его ареал обитания тянется вдоль северо-западного побережья Южной Америки и захватывает высокогорья Анд. Известные на языке кечуа как *Siwar q'inti*, эти колибри упоминаются в местном фольклоре как знак удачи.

Сверкающие колибри имеют переливчато-зеленый окрас с фиолетовыми отметинами на голове и груди. Более длинные пурпурные перья возле ушей вырастают у самцов в период токования. Эти птицы, довольно крупные для колибри, вырастают в среднем до 12–15 сантиметров в длину и весят около 8 граммов. Самки откладывают по два яйца в гнездо, которое свили сами, и высиживают их. Птенцы вылетают из гнезда в возрасте трех недель.

Обитая в холодных высокогорьях, сверкающие колибри относятся к видам колибри, которые каждую ночь впадают в глубокое оцепенение, чтобы заснуть. Это состояние похоже на зимнюю спячку и характеризуется снижением функций жизнедеятельности организма и почти полной акклиматизацией к низким температурам окружающей среды, которые меняются на рассвете. Благодаря такому состоянию эти колибри способны пережить долгие холодные ночи без пищи, которая в противном случае была бы необходима, чтобы не мерзнуть. Механизмы, лежащие в основе такого сложного умения птиц, являются предметом продолжающихся научных исследований.

Сверкающий колибри распространен во всем своем ареале и оценивается МСОП как вызывающий наименьшее беспокойство. Многие животные на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры из *«Естественной истории»* Томаса Вуда.

Ронни Митра, Иракли Надареишвили

Микросервисы. От архитектуры до релиза

Серия «Бестселлеры O'Reilly»

Перевел с английского С. Черников

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*Т. Никифорова, Е. Павлович*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 07.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 12.05.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 27,090. Тираж 700. Заказ 0000.

Сэм Ньюмен

СОЗДАНИЕ МИКРОСЕРВИСОВ

2-е издание

По мере того как организации переходят от монолитных приложений к небольшим автономным микросервисам, распределенные системы становятся все более детализированными. Второе дополненное издание предлагает целостный взгляд на самые актуальные темы, в которых необходимо разбираться при создании и масштабировании архитектуры микросервисов, а также управлении ею.

Вы познакомитесь с современными решениями для моделирования, интеграции, тестирования, развертывания и мониторинга собственных автономных сервисов. Примеры из реальной жизни показывают, как получить максимальную отдачу от этих архитектур. Книга будет полезна всем: от архитекторов и разработчиков до тестировщиков и специалистов по эксплуатации.

КУПИТЬ

Марк Ричардс, Нил Форд

ФУНДАМЕНТАЛЬНЫЙ ПОДХОД К ПРОГРАММНОЙ АРХИТЕКТУРЕ: ПАТТЕРНЫ, СВОЙСТВА, ПРОВЕРЕННЫЕ МЕТОДЫ

Архитекторы ПО стабильно входят в десятку самых высокооплачиваемых профессий. Но до сих пор не было реального руководства, которое позволило бы разработчикам стать архитекторами. И вот наконец появилась книга, в которой дается всеобъемлющий обзор разнообразных аспектов архитектуры программного обеспечения. Начинающие и уже состоявшиеся архитекторы найдут в ней паттерны архитектур, определения компонентов, приемы построения эволюционных архитектур и множество других тем.

Марк Ричардс и Нил Форд обладают бесценным практическим опытом, профессионально занимаются этой темой, уделяя особое внимание принципам построения архитектуры, применимым ко всем технологическим стекам. Они предлагают современный взгляд на архитектуру ПО с учетом всех нововведений последнего десятилетия.

КУПИТЬ