Тестирование микросервисов, часть 1

В этой главе

- Эффективные стратегии тестирования микросервисов.
- Применение макетов и заглушек для изолированного тестирования программных элементов.
- Использование пирамиды тестов для расстановки приоритетов при тестировании.
- Модульное тестирование классов внутри сервиса.

В FTGO, как и во многих других организациях, подход к тестированию традиционный. *Тестирование* — это процесс, проводимый в основном после разработки. Разработчики передают написанный код коллегам из отдела обеспечения качества, которые проверяют, работает ли программный продукт так, как задумано. Большая часть тестирования выполняется вручную. К сожалению, такой подход неприемлем по двум причинам.

- *Ручное тестирование чрезвычайно неэффективно.* Никогда не просите человека сделать то, что у компьютера получается намного лучше. По сравнению с компьютерами люди работают с низкой производительностью и не способны делать это круглосуточно. Если вы полагаетесь на ручное тестирование, забудьте о быстрой и безопасной доставке программного обеспечения. Написание автоматических тестов — это очень важно.
- *Тестирование производится на слишком позднем этапе процесса доставки.* Безусловно, тестирование уже написанного приложения имеет право на существование, но, как показывает опыт, этого недостаточно. Написание автоматических тестов

намного лучше сделать частью разработки. Это повысит продуктивность, поскольку разработчики смогут видеть результаты тестирования во время редактирования кода.

В этом отношении FTGO — типичная организация. Отчет Sauce Labs Testing Trends за 2018 год рисует довольно мрачную картину состояния автоматизации тестов (saucelabs.com/resources/white-papers/testing-trends-for-2018). В нем говорится, что в основном автоматизированными являются лишь 26 % организаций, а полностью автоматизированными — жалкие 3 %!

Зависимость от ручных тестов не связана с нехваткой инструментария и фреймворков. Например, JUnit — популярный фреймворк тестирования для Java — был выпущен еще в 1998 году. Плачевное состояние автоматических тестов вызвано культурными аспектами: «тестированием должен заниматься отдел обеспечения качества», «у разработчиков есть более важные задачи» и т. д. Ситуацию не улучшает и то, что создание набора быстрых, эффективных и легкоподдерживаемых тестов требует больших усилий. К тому же крупное монолитное приложение, как правило, очень сложно протестировать.

Как упоминалось в главе 2, ключевым фактором при выборе микросервисной архитектуры является улучшение тестируемости. В то же время из-за своей сложности микросервисный подход *требует* написания автоматических тестов. Более того, некоторые аспекты тестирования микросервисов трудно реализовать, ведь необходимо убедиться в том, что они могут корректно взаимодействовать между собой, но при этом минимизировать количество медленных, сложных и ненадежных сквозных тестов, которые требуют запуска множества сервисов.

Это первая из двух глав, посвященных тестированию. Считайте ее введением. В главе 10 рассматриваются более продвинутые концепции. Обе они довольно длинные, но в них вы сможете найти идеи и методики тестирования, необходимые для разработки современного программного обеспечения в целом и микросервисной архитектуры в частности.

Я начну главу с описания эффективных стратегий тестирования микросервисных приложений. Эти стратегии дадут вам уверенность в работоспособности вашего кода, минимизируя при этом сложность тестов и время их выполнения. После этого я продемонстрирую написание определенного вида тестов для сервисов, называемых модульными. Другие типы тестов — интеграционные, компонентные и сквозные — будут описаны в главе 10.

Поговорим о стратегиях тестирования микросервисов.

Почему мы начинаем с введения в тестирование

Вам, наверное, интересно, почему эта глава включает в себя знакомство с базовыми принципами тестирования. Если вы уже имели дело с такими понятиями, как пирамида тестов и тесты разных типов, можете быстро пролистать ее и перейти к следующей, которая посвящена аспектам тестирования, связанным с микросервисами. Но мой опыт консультации и обучения клиентов по всему миру говорит о том, что фундаментальной слабостью многих организаций, занимающихся разработкой программного обеспечения,

является отсутствие автоматических тестов. Автоматическое тестирование *необходимо*, если вам нужна быстрая и надежная доставка обновлений. Это единственный способ сокращения времени, которое уходит на развертывание зафиксированного кода в промышленной среде. Возможно, еще более важный момент — то, что автоматические тесты заставляют вас разрабатывать приложения, которые в принципе можно протестировать. Обычно автоматическое тестирование очень непросто внедрить в большой и сложный проект. Иными словами, если вы хотите максимально быстро очутиться в монолитном аду, не пишите автоматические тесты.

9.1. Стратегии тестирования микросервисных архитектур

Представьте, что вы вносите изменение в сервис `Order` приложения FTGO. Вслед за этим будет естественно запустить измененный код и убедиться в его корректной работе. Вы можете протестировать изменение вручную. Сначала нужно запустить сервис `Order` и все его зависимости, включая инфраструктурные компоненты наподобие БД и другие сервисы приложения. Затем, чтобы проверить сервис, вы должны обратиться к нему либо через его API, либо через пользовательский интерфейс приложения. Это медленный ручной способ тестирования кода.

Куда более подходящим выбором будет написание автоматических тестов, которые можно запускать во время разработки. Процесс написания приложения должен выглядеть так: отредактировать код, запустить тесты (в идеале одним нажатием клавиши), повторить. Если тесты выполняются быстро, через несколько секунд станет ясно, работает ли измененный код. Но как сделать тесты быстрыми? Как узнать, требуется ли более комплексное тестирование? Ответы на эти вопросы я даю в этом и следующем разделах.

Данный раздел начинается с обзора важных концепций автоматического тестирования. Вы увидите, какие задачи оно решает, и познакомитесь со структурой типичного теста. Я опишу различные типы тестов, которые вам нужно будет создавать. Также будет представлена пирамида тестов, которая служит полезным руководством относительно того, какие участки кода больше всего нуждаются в тестировании. После знакомства с основными концепциями мы поговорим о стратегиях тестирования микросервисов и присущих им конкретных трудностях. Вы познакомитесь с методиками, позволяющими писать более простые, быстрые, но в то же время эффективные тесты для микросервисной архитектуры.

Итак, начнем с концепций тестирования.

9.1.1. Обзор методик тестирования

В этой главе основное внимание уделяется автоматическим тестам. Упоминая здесь какие-либо *тесты*, я подразумеваю, что они *автоматические*. «Википедия» дает следующее определение *тестового случая*: *«Тестовый случай — это формально*

описанный алгоритм тестирования программы, специально созданный для определения возникновения в программе определенной ситуации, определенных выходных данных»[1].

Иными словами, целью теста (рис. 9.1) является проверка поведения тестируемой системы. Под *системой* здесь подразумевается элемент программного обеспечения, к которому применяется тест. Это может быть всего лишь класс, или целое приложение, или нечто среднее, например набор классов или отдельный сервис. Коллекция взаимосвязанных тестов формирует *тестовый набор*.

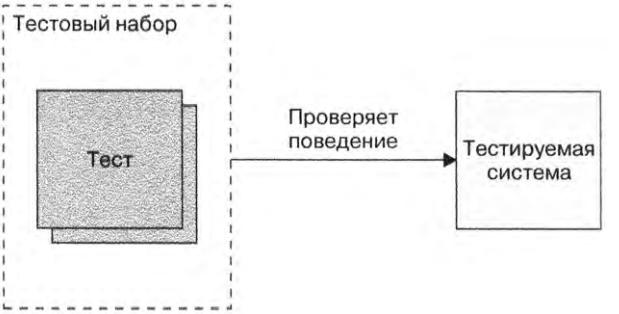

Рис. 9.1. Цель теста состоит в проверке поведения тестируемой системы. Система может быть всего лишь классом или целым приложением

Сначала мы рассмотрим концепцию автоматических тестов. Затем обсудим тесты разных типов, которые вам придется писать. После этого будет представлена пирамида тестов, описывающая относительные пропорции тестов, необходимых в том или ином случае.

Написание автоматических тестов

Автоматические тесты обычно пишут, используя специальный фреймворк. Например, JUnit — популярный фреймворк тестирования для Java. Структура автоматического теста показана на рис. 9.2. Каждый тест реализуется в виде метода, который принадлежит тестовому классу.

Типичный автоматический тест состоит из четырех этапов (xunitpatterns.com/Four%20Phase%20Test.html).

1. *Подготовка* — инициализирует среду тестирования, состоящую из самой системы и ее зависимостей, приводя ее в нужное состояние. Например, создает тестируемый класс и приводит его в состояние, необходимое для демонстрации желаемого поведения.
2. *Выполнение* — запускает тестируемую систему, например вызов метода из тестируемого класса.

[1] ru.wikipedia.org/wiki/Вариант_тестирования.

3. *Проверка* — делает выводы о результате выполнения и состоянии тестируемой системы. Например, проверяет значение, возвращаемое методом, и новое состояние тестируемого класса.
4. *Очистка* — удаляет среду тестирования, если это необходимо. Многие тесты пропускают этот этап, но, например, при тестировании БД иногда нужно откатить транзакции, инициированные на этапе подготовки.

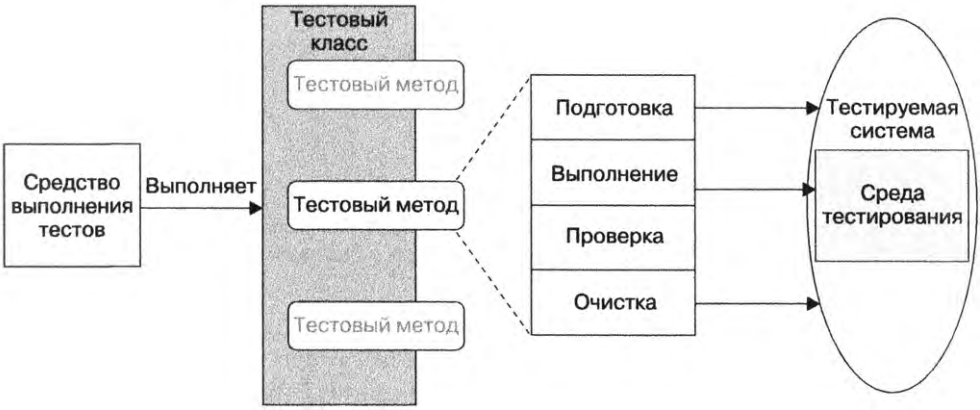

Рис. 9.2. Каждый автоматический тест реализуется тестовым методом, который принадлежит тестовому классу. Тест состоит из четырех этапов: подготовки — подготовки среды тестирования (то есть всего, что необходимо для выполнения теста); выполнения — запуска тестируемой системы; проверки — оценки результатов теста; очистки — удаления среды тестирования

Чтобы уменьшить дублирование кода и упростить тесты, в тестовый класс иногда добавляют подготовительные методы, которые выполняются перед вызовом самого теста, и методы очистки, реализуемые в самом конце. Тестовый *набор* — это перечень тестовых классов. Для их запуска используется *средство выполнения тестов*.

Тестирование с помощью макетов и заглушек

Тестируемая система часто имеет зависимости, которые могут осложнить и замедлить ваши тесты. Например, класс `OrderController` обращается к сервису `Order`, который так или иначе зависит от многих других прикладных и инфраструктурных сервисов. Тестирование класса `OrderController` путем запуска значительной части приложения было бы непрактичным. Нам нужно как-то изолировать свои тесты.

Решение, показанное на рис. 9.3, состоит в замене зависимостей тестируемой системы дублерами. *Дублер* — это объект, который симулирует поведение зависимости.

Существует два вида дублеров: заглушки (stubs) и макеты (mocks). Эти термины часто считают взаимозаменяемыми, хотя они немного различаются. *Заглушка* — это дублер, который возвращает значения тестируемой системе. *Макет* — это дублер, используемый тестом для проверки того, что тестируемая система корректно вызывает свои зависимости. Во многих случаях макет является заглушкой.

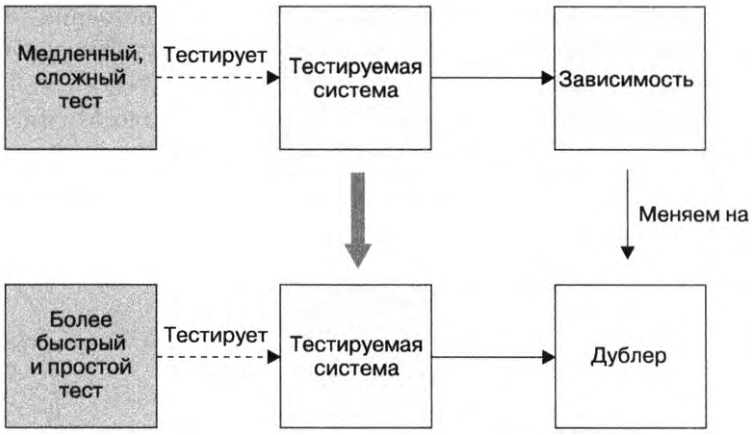

Рис. 9.3. Замена зависимости дублером, который позволяет тестировать систему в изоляции. Тест получается более простым и быстрым

Позже в этой главе вы увидите примеры применения дублеров. Так, в подразделе 9.2.5 будет показано, как протестировать класс `OrderController` в изоляции, задействуя дублер класса `OrderService`. В этом примере дублер `OrderService` реализуется с помощью Mockito — популярного фреймворка для создания макетов объектов в Java. В главе 10 вы увидите, как протестировать сервис `Order`, используя дублеры тех сервисов, к которым он обращается. Эти дублеры будут отвечать на командные сообщения, отправляемые сервисом `Order`.

Теперь рассмотрим разные типы тестов.

Типы тестов

Существует множество типов тестов. Некоторые из них, такие как тесты производительности и удобства использования, позволяют убедиться в том, что приложение отвечает требованиям к качеству обслуживания. В этой главе основное внимание уделяется автоматическим тестам, которые проверяют функциональные аспекты приложения или сервисов. Вы узнаете, как пишутся тесты четырех разных типов:

- *модульные тесты*, которые тестируют небольшую часть сервиса, такую как класс;
- *интеграционные тесты*, которые проверяют, может ли сервис взаимодействовать с инфраструктурными компонентами, такими как базы данных и другие сервисы приложения;
- *компонентные тесты* — приемочные тесты для отдельного сервиса;
- *сквозные тесты* — приемочные тесты для целого приложения.

Они различаются в основном охватом. На одном конце спектра находятся модульные тесты, которые проверяют поведение наименьшего значимого элемента

программы. В объектно-ориентированных языках, таких как Java, это класс. Их противоположность — сквозные тесты, проверяющие поведение целого приложения. Посередине находятся компонентные тесты, относящиеся к отдельным сервисам. Интеграционные тесты, как вы увидите в следующей главе, имеют относительно небольшой охват, но они сложнее, чем обычные модульные тесты. Охват — это лишь один из способов охарактеризовать тест. Еще одна характеристика — тестовый квадрант.

Модульные тесты на этапе компиляции

Тестирование — это неотъемлемая часть разработки. Современный процесс разработки состоит из редактирования кода с последующим запуском тестов. Более того, если вы практикуете TDD (Test-Driven Development — разработка через тестирование), создание новой функции или исправление ошибки подразумевает предварительное написание неисправного теста, который ваш код должен успешно пройти. Но даже если вы не поклонник TDD, написание теста, воспроизводящего ошибку, и кода, который ее исправляет, — превосходный подход.

Тесты, которые запускаются в ходе этого процесса, называются тестами *этапа компиляции*. В современных IDE, таких как IntelliJ IDEA и Eclipse, обычно не предусматривается отдельного шага для компиляции приложения. Вместо этого обеспечивается комбинация клавиш, которая компилирует код и сразу запускает тесты. Чтобы оставаться в этом потоке, тесты должны выполняться как можно быстрее — желательно не дольше нескольких секунд.

Использование тестового квадранта для классифицирования тестов

Хороший способ классифицировать тесты — тестовый квадрант, предложенный Брайаном Мариком (Brian Marick) (www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1). Он группирует тесты по двум основаниям (рис. 9.4).

- *К чему относится тест — к бизнесу или технологиям.* Бизнес-тест описывается в терминологии специалиста проблемной области, тогда как для описания технологического теста используется терминология разработчиков и реализации.
- *Какова цель теста — помочь с написанием кода или дать оценку приложению.* Разработчики применяют тесты, которые помогают им в ежедневной работе. Тесты, оценивающие приложение, нужны для определения проблемных участков.

Тестовый квадрант определяет четыре категории тестов:

- *Q1* — помощь в программировании с ориентацией на технологии — модульные и интеграционные тесты;
- *Q2* — помощь в программировании с ориентацией на бизнес — компонентные и сквозные тесты;

- *Q3* — оценка приложения с точки зрения бизнеса — проверка удобства использования и исследовательское тестирование;
- *Q4* — оценка приложения с точки зрения технологий — нефункциональное приемочное тестирование, такое как проверка производительности.

Помимо тестового квадранта, существуют и другие способы организации тестов. Например, пирамида тестов помогает определить, сколько тестов каждого типа следует написать.

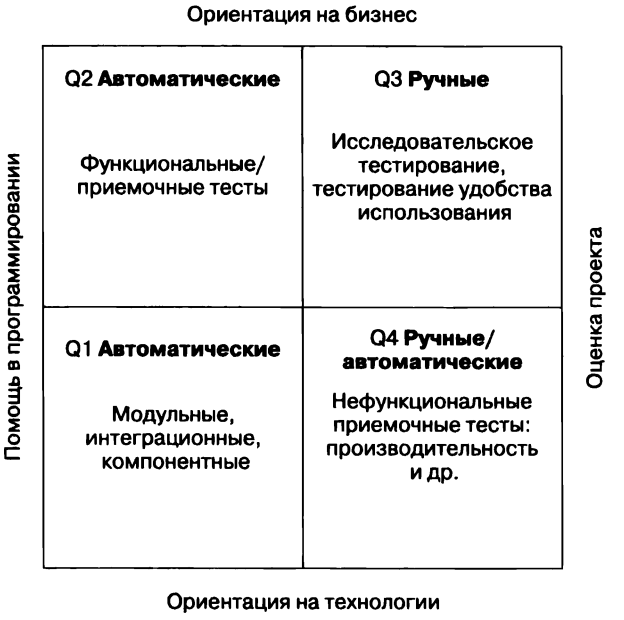

Рис. 9.4. Тестовый квадрант классифицирует тесты по двум основаниям. Первое — это ориентация теста на бизнес или технологии. Второе — назначение теста: помощь в программировании или оценка приложения

Применение пирамиды тестов как средства приоритизации тестирования

Чтобы удостовериться, что наше приложение работает, мы должны написать разного рода тесты. Но проблема в том, что с увеличением охвата теста растут его сложность и время выполнения. Кроме того, чем шире охват теста и чем больше составных элементов он в себя включает, тем менее надежным становится. Ненадежные тесты не намного лучше, чем их отсутствие, ведь, если тесту нельзя доверять, его сбои, скорее всего, будут игнорироваться.

На одном конце спектра располагаются модульные тесты для отдельных классов. Они надежны, просты в написании и быстро выполняются. На противоположном конце находятся сквозные тесты для всего приложения. Они медленные, их сложно

писать, а из-за сложности они часто оказываются ненадежными. Поскольку наш бюджет на разработку и тестирование ограничен, мы хотим сосредоточиться на написании тестов с небольшим охватом, не ставя при этом под угрозу эффективность тестового набора.

Хорошим подспорьем в этом может стать пирамида тестов (рис. 9.5) (martinfowler.com/bliki/TestPyramid.html). В ее основании лежат быстрые, простые и надежные тесты. Сквозные тесты, отличающиеся низкой скоростью, высокой сложностью и ненадежностью, расположены на вершине. Пирамида тестов описывает относительные пропорции каждого типа тестирования. Она похожа на пирамиду питания, которую публикует Министерство сельского хозяйства США (ru.wikipedia.org/wiki/Пирамида_питания), но, честно говоря, приносит больше пользы и вызывает меньше споров.

Рис. 9.5. Пирамида тестов описывает относительные пропорции типов тестов, которые нужно написать. Продвигаясь вверх, вы должны писать все меньше и меньше тестов

Суть этого подхода заключается в том, что с продвижением вверх по пирамиде мы должны писать все меньше и меньше тестов. Модульных тестов должно быть много, а сквозных — мало.

В этой главе я рассмотрю стратегию, которая делает акцент на тестировании элементов сервиса. Она минимизирует даже количество компонентных тестов, которые проверяют сервис целиком.

Тестирование отдельных микросервисов наподобие `Consumer`, не зависящих от других сервисов, не составляет труда. Но как насчет таких сервисов, как `Order`, у которых есть многочисленные зависимости? Как можно быть уверенным в том, что приложение в целом работоспособно? Это ключевые вопросы, которые относятся к тестированию приложений с микросервисной архитектурой. Сложность тестирования — характеристика не столько отдельных сервисов, сколько взаимодействия между ними. Давайте посмотрим, как подойти к этой проблеме.

9.1.2. Трудности тестирования микросервисов

Межпроцессное взаимодействие играет намного более важную роль в микросервисной архитектуре, чем в монолитном приложении. Монолитный код может общаться с несколькими внешними клиентами и сервисами. Например, монолитная версия FTGO использует несколько сторонних веб-сервисов со стабильными API: Stripe для платежей, Twilio для обмена сообщениями и Amazon SES — для почты. Любое взаимодействие между внутренними модулями происходит на уровне языка программирования. Фактически IPC находится на границе приложения.

Для сравнения: в микросервисной архитектуре межпроцессное взаимодействие играет одну из ключевых ролей. Микросервисное приложение — распределенная система. Разные команды заняты разработкой своих сервисов и развитием их API. Очень важно, чтобы разработчики сервиса писали тесты, которые проверяют, как он взаимодействует со своими зависимостями и клиентами.

Как говорилось в главе 3, сервисы могут общаться между собой, применяя разнообразные стили и механизмы IPC. В некоторых случаях используется стиль «запрос/ответ», который реализуется с помощью таких синхронных протоколов, как REST или gRPC. Сервисы могут общаться также в стиле «запрос/асинхронный ответ» или «издатель/подписчик», обмениваясь асинхронными сообщениями. Например, на рис. 9.6 показано, как взаимодействуют некоторые сервисы приложения FTGO. Каждая стрелка ведет от потребителя к отправителю.

Стрелка указывает в направлении зависимости — от потребителя API к сервису, который его предоставляет. Ожидания потребителя относительно API зависят от природы взаимодействия:

- *REST-клиент → сервис.* API-шлюз направляет запросы к сервисам и занимается объединением API.
- *Потребитель доменных событий → издатель.* Сервис `Order History` потребляет события, публикуемые сервисом `Order`.
- *Сторона, запрашивающая командные сообщения → отвечающая сторона.* Сервис `Order` шлет командные сообщения различным сервисам и потребляет их ответы.

Каждое взаимодействие между двумя сервисами может быть представлено в виде соглашения или контракта. Например, сервисы `Order History` и `Order` должны согласовать структуру сообщений с событиями и канал, в который те будут публиковаться. Точно так же API-шлюз и сервисы должны согласовать конечные точки REST API. К тому же сервис `Order` и все другие сервисы, с которыми он общается с помощью асинхронных запросов и ответов, должны выбрать командный канал, а также формат команд и ответов.

Вы, как разработчик сервиса, должны быть уверены в стабильности API, которые потребляете. По этой же причине не следует вносить ломающие изменения в API собственного сервиса. Например, если вы работаете над сервисом `Order`, то должны быть уверены в том, что разработчики его зависимостей, таких как сервисы `Consumer` и `Kitchen`, не нарушат совместимость их API с вашим кодом. Точно так же вы должны следить за тем, чтобы изменения в API сервиса `Order` не повлияли на совместимость с API-шлюзом или сервисом `Order History`.

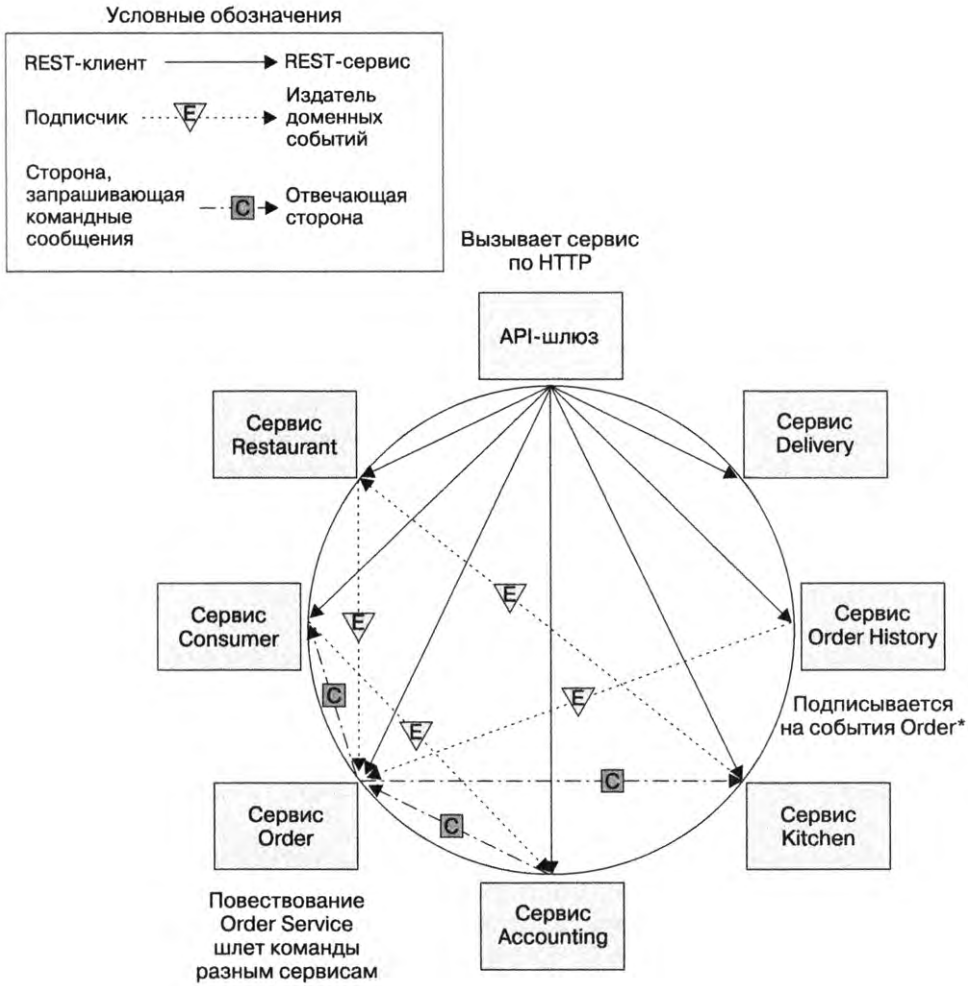

Рис. 9.6. Часть межсервисного взаимодействия в приложении FTGO

Чтобы проверить, способны ли два сервиса общаться между собой, можно обратиться к API, который инициирует взаимодействие, и убедиться в том, что он возвращает ожидаемый результат. Это точно поможет выявить проблемы с интеграцией, но это, в сущности, сквозной тест. Скорее всего, ему придется запускать множество других переходных зависимостей этих сервисов. Возможно, он также должен будет вызывать сложные высокоуровневые функции, такие как бизнес-логика, хотя его цель — проверка относительно низкоуровневого механизма IPC. Лучше всего избегать написания подобных сквозных тестов. Нам нужны более быстрые, простые и надежные тесты, которые в идеале проверяют работу сервисов в изоляции. В качестве решения можно воспользоваться так называемым *тестированием контрактов с расчетом на потребителя*.

Тестирование контрактов с расчетом на потребителя

Представьте, что вы являетесь членом команды, которая занимается разработкой API-шлюза, описанного в главе 8. Объект шлюза `OrderServiceProxy` обращается к различным конечным точкам REST, включая `GET /orders/{orderId}`. В этом случае крайне важно иметь тесты, которые проверяют согласованность API между API-шлюзом и сервисом `Order`. В терминологии тестирования потребительских контрактов между этими двумя сервисами имеется связь *«потребитель — провайдер»*. Потребителем выступает API-шлюз, а провайдером — сервис `Order`. Проверка потребительского контракта — это интеграционный тест для провайдера, такого как сервис `Order`, он позволяет убедиться в том, что API провайдера отвечает ожиданиям потребителя, такого как API-шлюз.

Тестирование потребительского контракта сосредоточено на проверке того, что API провайдера по своей форме отвечает ожиданиям потребителя. В данном случае оно позволяет убедиться в том, что провайдер реализует конечную точку REST, которая:

- имеет нужные HTTP-метод и путь;
- принимает нужные заголовки, если таковые имеются;
- принимает тело запроса, если оно имеется;
- возвращает ответ с ожидаемыми кодом состояния, заголовками и телом.

Следует помнить, что тесты контрактов не занимаются тщательной проверкой бизнес-логики провайдера. За это отвечают модульные тесты. Позже вы увидите, что в контексте REST API тесты потребительских контрактов на самом деле являются тестами макетов контроллеров.

Команда, разрабатывающая потребительский код, пишет набор тестов для контрактов и делает его частью тестового набора провайдера, например, через запрос на принятие изменений. Разработчики других сервисов, которые обращаются к сервису `Order`, тоже вносят свой вклад в этот набор (рис. 9.7). Каждый набор тестов будет проверять те аспекты API `Order`, которые относятся к тому или иному потребителю. Например, тестовый набор для сервиса `Order History` проверяет, публикует ли сервис `Order` ожидаемые события.

Эти тестовые наборы выполняются в процессе развертывания сервиса `Order`. Если проверка потребительского контракта завершается неудачно, разработчики провайдера делают вывод о том, что они внесли ломающее изменение в API. Им следует либо исправить API, либо связаться с командой потребительской стороны.

Шаблон «Тестирование контрактов с расчетом на потребителя»

Проверяет, соответствует ли сервис ожиданиям своих клиентов. См. microservices.io/patterns/testing/service-integration-contract-test.html.

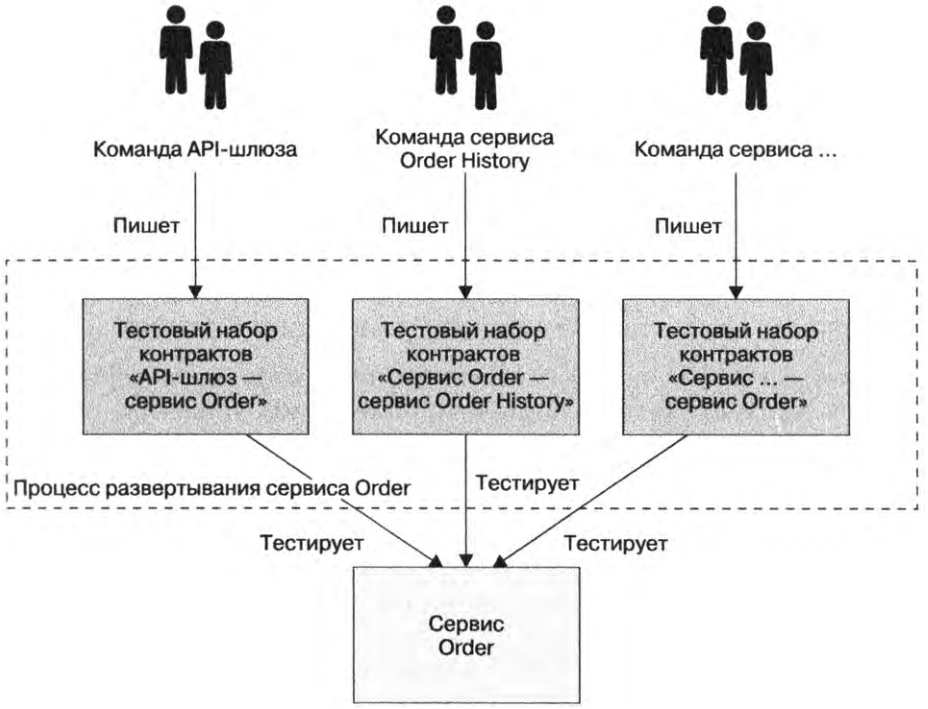

Рис. 9.7. Команда каждого сервиса, который потребляет API сервиса Order, предоставляет тестовый набор контрактов. Этот набор проверяет, соответствует ли API ожиданиям потребителей. Данные тесты в сочетании с тестовыми наборами других команд выполняются в рамках процесса развертывания сервиса Order

Тесты контрактов с расчетом на потребителя обычно применяют тестирование по примеру. Взаимодействие между потребителем и провайдером определяется набором примеров, которые называются контрактами. Каждый *контракт* состоит из примеров сообщений, обмен которыми происходит во время взаимодействия. Скажем, контракт для REST API состоит из примеров HTTP-запроса и ответа. На первый взгляд может показаться, что взаимодействие лучше описывать в виде схем в формате OpenAPI или JSON. Но, как оказалось, схемы не очень подходят для написания тестов. Они могут помочь с проверкой ответа, но тест все равно должен обратиться к провайдеру с примером запроса.

Более того, потребительским тестам нужны еще и примеры ответов. Несмотря на то что основной задачей данного подхода является проверка провайдера, контракты также проверяют, соответствует ли им потребитель. Например, потребительский контракт для REST-клиента конфигурирует заглушку сервиса, которая проверяет, совпадает ли HTTP-запрос с запросом контракта, и возвращает обратно его HTTP-ответ. Тестирование обеих сторон взаимодействия позволяет убедиться в том, что потребитель и провайдер согласовали API. Позже вы увидите примеры написания

подобного рода тестов, но сначала посмотрим, как выполнять тестирование потребительских контрактов с помощью Spring Cloud Contract.

Шаблон «Тестирование контрактов на стороне потребителя»

Проверяет, может ли клиент взаимодействовать с сервисом. См. microservices.io/patterns/testing/consumer-side-contract-test.html.

Тестирование сервисов с помощью Spring Cloud Contract

Существует два популярных фреймворка для тестирования контрактов: Spring Cloud Contract (https://spring.io/projects/spring-cloud-contract), который позволяет тестировать потребительские контракты в приложениях, основанных на Spring, и семейство фреймворков Pact (github.com/pact-foundation) с поддержкой разных языков. Приложение FTGO использует фреймворк Spring, поэтому в данной главе я покажу, как работать со Spring Cloud Contract. Для написания контрактов эта технология предоставляет проблемно-ориентированный язык (DSL) в стиле Groovy. Каждый контракт — это конкретный пример взаимодействия между потребителем и провайдером, такой как HTTP-запрос и ответ. Код Spring Cloud Contract генерирует тесты контрактов для провайдера. Он также настраивает макеты (например, макет HTTP-сервера) для потребительских интеграционных тестов.

Представьте, к примеру, что вы работаете над API-шлюзом и хотите написать тест потребительского контракта для сервиса `Order`. Этот процесс (рис. 9.8) подразумевает взаимодействие с командами, отвечающими за этот сервис. Вы пишете контракты, которые определяют, как API-шлюз общается с сервисом `Order`. Команда сервиса `Order` использует эти контракты для тестирования своего сервиса, а вы с их помощью проверяете свой API-шлюз. Последовательность шагов приведена далее.

1. Вы пишете один или несколько контрактов (пример приведен в листинге 9.1). Каждый контракт состоит из HTTP-запроса, который API-шлюз может послать сервису `Order`, и ожидаемого HTTP-ответа. Эти контракты вы передаете команде сервиса `Order` (возможно, посредством запроса на принятие изменений в Git).
2. Команда сервиса `Order` тестирует его с помощью тестов, код которых сгенерирован из потребительских контрактов с помощью Spring Cloud Contract.
3. Команда сервиса `Order` публикует полученные контракты в репозиторий Maven.
4. Вы используете опубликованные контракты, чтобы написать тесты для API-шлюза.

Поскольку вы тестируете API-шлюз с помощью опубликованных контрактов, то можете быть уверены в его совместимости с развернутым сервисом `Order`.

Контракты — это ключевая часть стратегии тестирования. В листинге 9.1 показан пример контракта для Spring Cloud Contract. Он состоит из HTTP-запроса и HTTP-ответа.

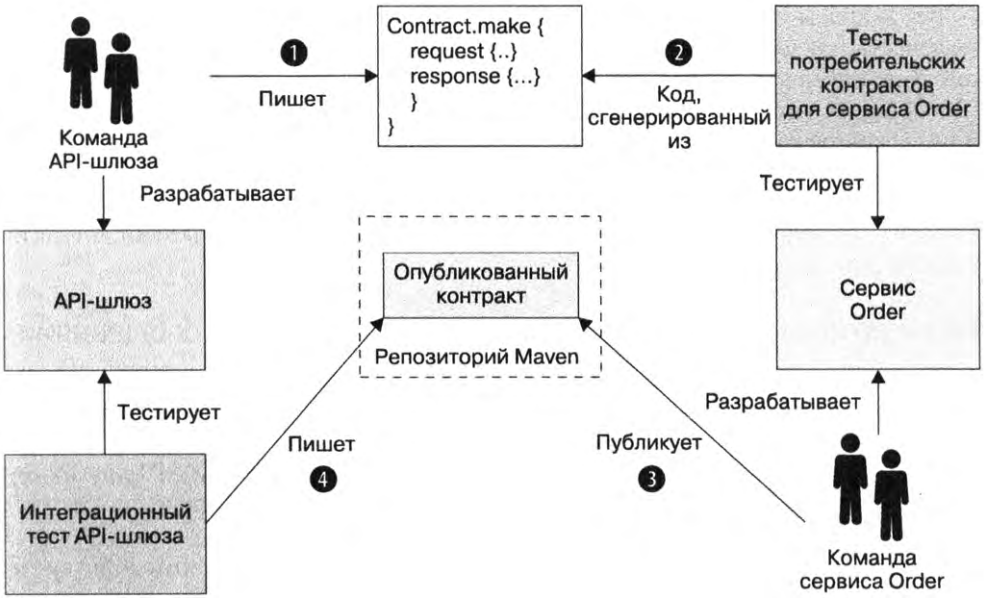

Рис. 9.8. Команда API-шлюза пишет контракты. Команда сервиса Order тестирует его с помощью этих контрактов и публикует их в репозиторий. Команда API-шлюза применяет опубликованные контракты для тестирования своего кода

Листинг 9.1. Контракт, описывающий то, как API-шлюз обращается к сервису Order

```
org.springframework.cloud.contract.spec.Contract.make {
    request {                          ← Метод и путь HTTP-запроса
        method 'GET'
        url '/orders/1223232'
    }
    response {                         ← Код состояния, заголовки
        status 200                       и тело HTTP-ответа
        headers {
            header('Content-Type': 'application/json;charset=UTF-8')
        }
        body("{ ... }")
    }
}
```

Роль запрашивающего элемента играет HTTP-запрос для конечной точки REST `GET /orders/{orderId}`. Ответный элемент представляет собой HTTP-ответ, описывающий заказ, ожидаемый API-шлюзом. Контракты на языке Groovy являются частью кодовой базы провайдера. Каждая потребительская команда создает контракты, которые описывают взаимодействие ее сервиса с провайдером, и передает их команде провайдера (возможно, через запрос принятия изменений в Git). Команда провайдера упаковывает контракты в архив JAR и публикует их в репозитории Maven. Тесты на стороне потребителя загружают этот архив из репозитория.

Все запросы и ответы контракта используются не только для тестирования, но и в качестве спецификации ожидаемого поведения. В тестах на стороне потребителя контракт применяется для конфигурации заглушки, аналогичной объекту-макету в Mockito, и симулирует поведение сервиса `Order`. Это позволяет тестировать API-шлюз без запуска этого сервиса. На стороне провайдера сгенерированный тестовый класс шлет провайдеру запрос контракта и проверяет, совпадает ли его ответ с ответом контракта. Подробно о Spring Cloud Contract мы поговорим в следующей главе, а пока что посмотрим, как использовать тестирование потребительских контрактов для API обмена сообщениями.

Тесты потребительских контрактов для API обмена сообщениями

REST-клиент — это не единственный вид потребителей с определенными ожиданиями относительно API провайдера. Потребителями могут выступать также сервисы, которые подписываются на доменные события и взаимодействуют с помощью асинхронных запросов/ответов. Они обращаются к асинхронным API других сервисов, делая предположения о природе этих API. Для них тоже нужно писать тесты потребительских контрактов.

Spring Cloud Contract также поддерживает тестирование взаимодействия на основе обмена сообщениями. Структура контракта и то, как он применяется в тестах, зависит от типа взаимодействия. Контракт для публикации доменных событий состоит из примера доменного события. В ходе тестирования провайдер генерирует событие и проверяет, совпадает ли оно с событием контракта. Потребительский тест проверяет, может ли потребитель обработать событие. Пример такого теста будет представлен в следующей главе.

Контракт для асинхронного взаимодействия в стиле «запрос/ответ» похож на HTTP-контракты. Он состоит из двух сообщений: с запросом и ответом. Тест провайдера шлет запрос контракта интерфейсу (API) и проверяет, совпадает ли полученный ответ с ответом контракта. Потребительский тест использует контракт, чтобы сконфигурировать заглушку для подписчика, которая перехватывает запрос контракта и возвращает указанный ответ. Пример такого теста описывается в следующей главе. Здесь же мы рассмотрим процесс развертывания, в рамках которого выполняются эти и другие тесты.

9.1.3. Процесс развертывания

У каждого сервиса есть свой процесс развертывания. В книге Джеза Хамбла (Jez Humble) *Continuous Delivery* (Addison-Wesley, 2010)[1] *процесс развертывания* описывается как автоматическая доставка кода из компьютера разработчика в промышленную среду. Он состоит из поэтапного выполнения тестов, вслед за которым происходит выпуск или развертывание сервиса (рис. 9.9). В идеале этот процесс должен быть полностью автоматизированным, но в реальности он может требовать ручного вмешательства. Процесс развертывания часто реализуется с помощью CI-сервера (Continuous Integration — непрерывное развертывание), такого как Jenkins.

[1] *Хамбл Д.* Непрерывное развертывание. — М.: Вильямс, 2011.

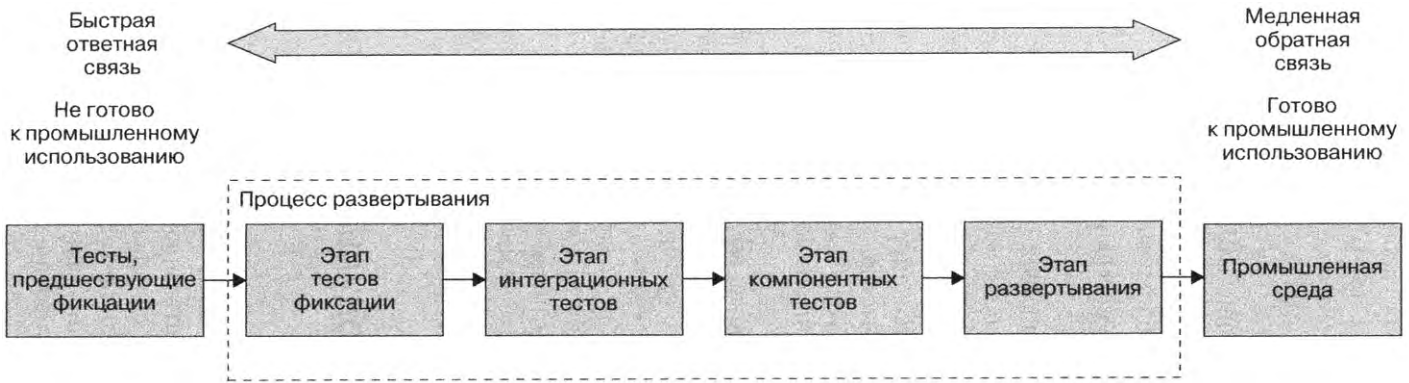

Рис. 9.9. Пример процесса развертывания для сервиса Order. Он состоит из последовательных этапов. Тесты, предшествующие фиксации, разработчик выполняет перед фиксацией кода. Остальные этапы реализует автоматизированная система, такая как CI-сервер Jenkins

По мере прохождения кода через процесс развертывания наборы тестов все более тщательно тестируют его в среде, приближенной к промышленной. Одновременно время выполнения каждого тестового набора обычно увеличивается. Основной смысл процедуры состоит в том, чтобы как можно скорее сообщить о непройденных тестах.

Процесс развертывания (см. рис. 9.9) состоит из следующих этапов.

- *Этап, предшествующий фиксации*, — выполняет модульные тесты. Запускается разработчиком перед фиксацией изменений.
- *Этап фиксации* — компилирует сервис, выполняет модульные тесты и производит статический анализ кода.
- *Интеграционный этап* — выполняет интеграционные тесты.
- *Компонентный этап* — выполняет компонентные тесты для сервиса.
- *Этап развертывания* — развертывает сервис в промышленной среде.

CI-сервер запускает этап фиксации, когда разработчик фиксирует изменение. Он выполняется чрезвычайно быстро, чтобы сразу предоставить сведения о фиксации. Дальнейшие этапы протекают дольше и предоставляют информацию не так быстро. Если все тесты пройдены, на заключительном этапе код развертывается в промышленную среду.

В этом примере автоматизирован весь процесс, от фиксации до развертывания. Однако в некоторых ситуациях требуется вмешательство человека. Например, вам может понадобиться этап ручного тестирования в предпромышленной среде. В этом сценарии код переходит на следующий этап, когда тестировщик отмечает успешное тестирование нажатием кнопки. Это может быть также выпуск новой версии сервиса в рамках процесса развертывания. Позже выпущенные сервисы будут упакованы и в качестве готового продукта отправлены заказчикам.

Теперь вы знаете, как организован процесс развертывания и на каких этапах выполняются разные типы тестов. Переместимся в самый низ пирамиды тестирования и посмотрим, как пишут модульные тесты для сервиса.

9.2. Написание модульных тестов для сервиса

Представьте, что вам нужно написать тест, который проверяет, вычисляет ли сервис `Order` приложения FTGO корректную промежуточную стоимость заказа. Код вашего теста может запустить сервис `Order`, обратиться к его REST API, чтобы создать заказ, и проверить, содержит ли HTTP-ответ ожидаемые значения. Однако при этом тест получится не только сложным, но и медленным. Если он выполняется на этапе компиляции класса `Order`, вы будете тратить много времени в ожидании его завершения. Написание модульных тестов для класса `Order` — куда более продуктивный подход.