1 Что такое API микросервисов

В этой главе

- ✓ Что такое микросервисы и чем они отличаются от монолитных приложений.
- ✓ Что такое веб-API и как они помогают осуществлять интеграцию между микросервисами.
- ✓ Наиболее важные задачи при разработке и эксплуатации микросервисов.

В этой главе дается определение наиболее важных понятий книги: микросервисов и API. *Микросервисы* — это архитектурный стиль, в котором компоненты системы проектируются как независимо развертываемые сервисы, а *API* — интерфейсы, позволяющие нам взаимодействовать с этими сервисами. Мы рассмотрим важные особенности микросервисной архитектуры и сравним ее с архитектурой монолитных приложений, которые строятся на основе единой кодовой базы и развертываются в виде единой сборки.

Мы обсудим преимущества и недостатки микросервисной архитектуры. В конце главы речь пойдет о проблемах, которые возникают при проектировании, внедрении и эксплуатации микросервисов. Все это я расскажу не для того, чтобы отговорить вас от внедрения микросервисов, а для того, чтобы вы могли принять взвешенное решение, подходят ли микросервисы именно для вас.

1.1. ЧТО ТАКОЕ МИКРОСЕРВИСЫ

В этом разделе мы определим, что такое микросервисная архитектура, и проанализируем, чем микросервисы отличаются от монолитных приложений. Рассмотрим преимущества и недостатки каждого архитектурного стиля. Наконец, мы также бегло вспомним исторические события, которые привели к появлению современной микросервисной архитектуры.

1.1.1. Определение микросервисов

Итак, что же такое микросервисы? Микросервисы можно определить по-разному, и в зависимости от того, какой аспект архитектуры мы хотим подчеркнуть, авторы дают немного различающиеся, но связанные между собой определения этого термина. Сэм Ньюман, один из самых известных авторов, пишущих о микросервисах, дает лаконичное определение: «Микросервисы — это небольшие, автономные, совместно работающие сервисы»[1].

Определение подчеркивает тот факт, что микросервисы — это приложения, которые работают независимо друг от друга, но могут взаимодействовать при выполнении своих задач. В определении также подчеркивается, что микросервисы «небольшие». В данном контексте «небольшой» относится не к размеру кодовой базы, а к идее, что микросервисы — это приложения с узкой и четко определенной областью применения, следующие принципу единственной ответственности — делать что-то одно, причем хорошо.

В основополагающей статье Джеймса Льюиса и Мартина Фаулера дается более подробное определение. Они характеризуют микросервисы как архитектурный стиль с «подходом к разработке одного приложения в виде набора небольших сервисов, каждый из которых выполняется отдельно и которые взаимодействуют с помощью упрощенных механизмов, часто с использованием ресурсов HTTP API» (https://martinfowler.com/articles/microservices.html). Это определение подчеркивает автономность сервисов, утверждая, что они выполняются в виде отдельных процессов. Льюис и Фаулер также отмечают, что микросервисы имеют узкий функционал, говоря, что они небольшие, и четко описывают, как микросервисы обмениваются информацией с помощью таких протоколов, как HTTP.

ОПРЕДЕЛЕНИЕ

Микросервис — это архитектурный стиль, в котором компоненты системы проектируются как независимо развертываемые сервисы. Микросервисы проектируются под четко обозначенные бизнес-функции в определенных модулях (поддоменах) и взаимодействуют друг с другом с помощью простых протоколов, таких как HTTP.

---

[1] *Ньюман С.* Создание микросервисов. — СПб.: Питер, 2016. — С. 23.

Из приведенных определений видно, что микросервисы можно охарактеризовать как архитектурный стиль, в котором сервисы — это компоненты, выполняющие небольшой и четко определенный набор связанных функций. Как видно на рис. 1.1, это означает, что микросервис проектируется и строится вокруг конкретной бизнес-задачи, например обработки платежей, отправки электронной почты или обработки заказов от покупателей.

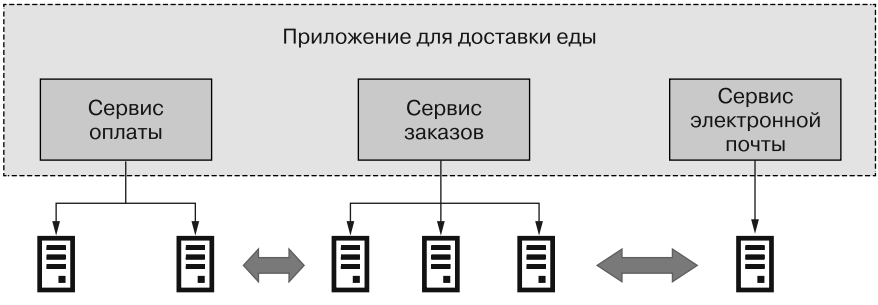

Рис. 1.1. В микросервисной архитектуре каждый сервис выполняет определенную бизнес-функцию и развертывается как независимый компонент, который выполняется в собственном процессе

Микросервисы развертываются как независимые процессы, обычно работающие в независимых средах, и предоставляют свой функционал через четко определенные интерфейсы. В книге вы научитесь создавать микросервисы, которые работают через веб-API, хотя возможны и другие типы интерфейсов, например очереди сообщений[1].

1.1.2. Микросервисы против монолитов

Теперь, когда мы знаем, что такое микросервисы, посмотрим, как они соотносятся с монолитной архитектурой приложений. В отличие от микросервисов монолит — это система, в которой вся функциональность развертывается в виде единой сборки и выполняется в одном процессе. Например, на рис. 1.2 показано приложение для доставки еды с четырьмя сервисами: оплаты, заказов, доставки и поддержки покупателей. Поскольку приложение реализовано как монолит, все функциональные возможности развертываются вместе. Мы можем запустить несколько экземпляров монолитного приложения и заставить их работать параллельно для обеспечения избыточности и масштабируемости, но в каждом процессе все равно будет работать целое приложение.

[1] Полный обзор различных интерфейсов, которые можно использовать для обеспечения взаимодействия между микросервисами, приведен в книге: *Ричардсон К.* Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019.

Рис. 1.2. В монолитном приложении вся функциональность развертывается вместе в виде единой сборки на каждом сервере

ОПРЕДЕЛЕНИЕ

Монолит — это архитектурный паттерн, в котором все приложение развертывается в виде единой сборки.

Монолит стоит выбирать, когда кодовая база невелика и предполагается, что она сильно не разрастется[1]. Монолиты имеют свои преимущества. Во-первых, наличие всей реализации в единой кодовой базе облегчает доступ к данным и бизнес-функциям из различных модулей. А поскольку все выполняется в рамках одного процесса, легко отследить ошибки в приложении: достаточно добавить несколько точек останова в разных частях кода, и вы получите подробную картину того, что происходит, когда что-то идет не так. Кроме того, поскольку весь код находится в рамках одного проекта, вы можете использовать инструменты повышения производительности в своей любимой среде разработки при работе с бизнес-логикой приложения из различных модулей.

[1] Подробный анализ стратегических архитектурных решений, связанных с монолитами и микросервисами, приведен в книге: *Vernon V., Jaskula T.* Strategic Monoliths and Microservices (Addison-Wesley, 2021).

Однако по мере роста и усложнения приложения у данного типа архитектуры проявляются ограничения. Это происходит, когда кодовая база увеличивается до такой степени, что ею становится трудно управлять, а поиск в коде оказывается сложной задачей. Кроме того, возможность повторного использования кода из других модулей в рамках одного проекта часто приводит к сильной связанности (tight coupling) между компонентами. Сильная связанность возникает, когда компонент зависит от деталей реализации другого фрагмента кода.

Чем больше монолит, тем больше времени требуется для его тестирования. Каждую часть монолита необходимо протестировать, и по мере добавления новых функций набор тестов увеличивается. Следовательно, развертывание становится медленнее и разработчикам приходится накапливать изменения в рамках одного релиза, что делает релизы более сложными. Поскольку многие изменения выпускаются вместе, то, если в релизе появляется новая ошибка, часто бывает трудно определить, какое именно изменение ее вызвало, и откатить его назад. А поскольку все приложение работает в рамках одного процесса, при масштабировании ресурсов для одного компонента требуется масштабирование всего приложения. Короче говоря, изменения кода оказываются все более рискованными, а управлять развертыванием становится все сложнее. Как микросервисы могут помочь решить эти проблемы?

Микросервисы помогают нам благодаря жестким границам, разделяющим компоненты. Когда вы реализуете приложение с использованием микросервисов, каждый микросервис работает в отдельном процессе (часто на разных серверах или виртуальных машинах) и может иметь любую модель развертывания. Фактически микросервисы могут быть написаны на совершенно разных языках программирования (но это не значит, что так и должно быть!).

Поскольку у микросервисов кодовая база меньше, чем у монолита, и их логика ограничена и определена в рамках конкретной бизнес-задачи, их легче тестировать и их наборы тестов выполняются быстрее. Микросервисы не зависят от других компонентов платформы на уровне кода (за исключением, возможно, некоторых общих библиотек), их код понятнее, и их легче рефакторить. Это означает, что со временем код может улучшаться и становиться более удобным для поддержки. Следовательно, мы можем вносить небольшие изменения в код и выпускать его чаще. Маленькие релизы более управляемы, и, если обнаружится ошибка, их легче откатить назад. Но я хотел бы подчеркнуть, что микросервисы — это не панацея. Как вы увидите в разделе 1.3, они также имеют ограничения и создают свои проблемы.

Теперь, разобравшись, что такое микросервисы и чем они отличаются от монолитных приложений, сделаем шаг назад и посмотрим, какие события привели к появлению этого типа архитектуры.

1.1.3. История появления современных микросервисов

Во многих отношениях микросервисы не новинка[1]. Компании внедряли и развертывали компоненты в виде независимых приложений задолго до того, как стала популярной концепция микросервисов. Они просто не называли это микросервисами. Вернер Фогельс, технический директор компании Amazon, рассказывает, что Amazon начала экспериментировать с этим типом архитектуры в начале 2000-х годов. К тому времени кодовая база сайта Amazon превратилась в сложную систему без четкой архитектурной схемы, а выпуск новых релизов и масштабирование системы стали серьезными проблемами. Для их решения они начали искать независимые части логики в коде и разделять их на независимо развертываемые компоненты, сопровождая их API. В рамках этого процесса в компании также определили данные, принадлежащие этим компонентам, и убедились, что другие части системы не могут получить доступ к этим данным иначе как через API. Они назвали этот новый тип архитектуры *сервис-ориентированной архитектурой* (https://vimeo.com/29719577). Компания Netflix также стала пионером в создании архитектурного стиля такого масштаба, и они назвали его «детальной сервис-ориентированной архитектурой» (fine-grained service oriented architecture) [2].

Для описания этого типа архитектуры термин *«микросервис»* стали активно употреблять в начале 2010-х годов. Например, Джеймс Льюис использовал это понятие на конференции в Кракове в 2012 году в презентации под названием *Micro-Services — Java, the Unix way* («Микросервисы — Java, путь Unix») (https://vimeo.com/74452550). В 2014 году концепция была закреплена статьей Мартина Фаулера и Джеймса Льюиса об архитектурных особенностях микросервисов (https://martinfowler.com/articles/microservices.html), а также публикацией известной книги Сэма Ньюмана «Создание микросервисов».

Сегодня микросервисы — это широко распространенный архитектурный стиль. Очень многие компании уже используют микросервисы или движутся в направлении их внедрения. Однако микросервисы подходят не всем и, несмотря на множество существенных преимуществ, также могут привести к значительным проблемам, о чем вы узнаете в разделе 1.3.

---

[1] Более полный анализ истории появления микросервисной архитектуры и ее предшественников см. в книге: *Dragoni N. et al.* Microservices: Yesterday, Today and Tomorrow, Present and Ulterior Software Engineering. — Springer, 2017. — P. 195–216.

[2] *Wang A., Tonse S.* Announcing Ribbon: Tying the Netflix Mid-Tier Services Together («Анонс ленты: связывая вместе сервисы промежуточного слоя Netflix») // Netflix Technology Blog, 18 января 2013 г. https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62. Прекрасное объяснение различий между сервис-ориентированной архитектурой (SOA) и микросервисной архитектурой см. в книге: *Ричардсон К.* Микросервисы. — С. 40–41.

1.2. ЧТО ТАКОЕ ВЕБ-API

В этом разделе мы поговорим о веб-API. Вы узнаете, что это конкретный пример более общего интерфейса прикладного программирования (API). Важно понимать, что API — это просто слой поверх приложения и что существуют различные типы интерфейсов. Начнем с определения API, а затем обсудим, как API позволяют нам интегрировать микросервисы.

1.2.1. API

API — это интерфейс, который позволяет нам программно взаимодействовать с приложением. Программные интерфейсы мы можем использовать из нашего кода или терминала — в этом их отличие от графических, в которых пользовательский интерфейс предназначен для взаимодействия с приложением. Существует несколько типов интерфейсов приложений: интерфейсы командной строки (CLI; позволяют использовать приложение из терминала), пользовательские интерфейсы (UI) для настольных компьютеров, пользовательские интерфейсы (UI) для веб-приложений или веб-API. Как показано на рис. 1.3, приложение может иметь один или несколько таких интерфейсов.

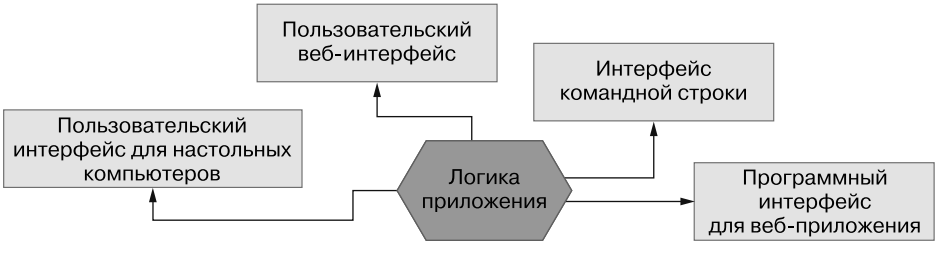

Рис. 1.3. Приложение может иметь несколько интерфейсов

Для иллюстрации этой идеи вспомните популярную утилиту cURL (client URL). cURL — это интерфейс командной строки для библиотеки `libcurl`, которая реализует функциональность, позволяющую нам взаимодействовать с URL-адресами, в то время как cURL раскрывает эти возможности через CLI. Например, мы можем использовать cURL для отправки GET-запроса на URL:

```
$ curl -L http://www.google.com
```

Мы также можем использовать cURL с флагом `-O`, чтобы загрузить содержимое URL в файл:

```
$ curl -O http://www.gnu.org/software/gettext/manual/gettext.html
```

Библиотека `libcurl` скрыта за интерфейсом командной строки cURL, и ничто не мешает нам получить к ней прямой доступ через исходный код (если вам интересно, можете взять его с Github: https://github.com/curl/curl) и создать для нее дополнительные типы интерфейсов.

1.2.2. Веб-API

Теперь, когда мы поняли, что такое API, перейдем к описанию веб-API. Веб-API — это API, который для передачи данных использует HyperText Transfer Protocol (HTTP). HTTP — это протокол связи, лежащий в основе Интернета и позволяющий нам обмениваться по сети различными типами данных (текстом, изображениями, видео и данными в формате JSON[1]). HTTP использует понятие Унифицированного указателя ресурса (Uniform Resource Locator, URL) для определения ресурсов в Интернете, а также предоставляет возможности, которые могут быть использованы при разработке API для улучшения взаимодействия с сервером, например методы запроса (GET, POST, PUT) и HTTP-заголовки. Веб-API реализуются с помощью таких технологий, как SOAP, REST, GraphQL, gRPC и других, которые подробно рассматриваются в приложении А.

1.2.3. Как API помогают нам управлять интеграцией микросервисов

Микросервисы взаимодействуют друг с другом через API, поэтому API по сути являются интерфейсами к микросервисам. Эти интерфейсы документируются с использованием стандартных протоколов. Документация API точно указывает нам, что нужно сделать для взаимодействия с микросервисом и каких ответов мы можем от него ожидать. Чем лучше документирован API, тем понятнее разработчикам клиентских приложений, как он работает. В этом смысле документацию API можно представить как контракт между сервисами (рис. 1.4).

Фаулер и Льюис популяризировали идею о том, что наилучшей стратегией интеграции микросервисов является создание *умных эндпоинтов (smart endpoints)* и взаимодействие через *глупые каналы (dumb pipes)* (https://martinfowler.com/articles/microservices.html). В основе этой идеи лежат принципы проектирования Unix-систем, которые устанавливают, что:

- система должна состоять из небольших независимых компонентов, которые выполняют только одну задачу;
- выходные данные для каждого компонента должны быть спроектированы таким образом, чтобы они могли легко стать входными данными для другого компонента.

[1] Данные в формате JSON в общем виде тоже текст. — *Примеч. ред.*

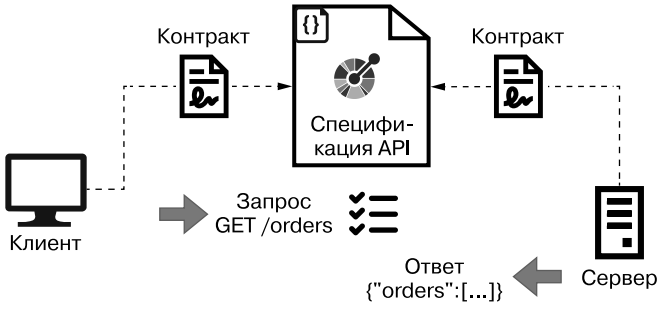

Рис. 1.4. Спецификация API представляет собой контракт между сервером и клиентом. Пока клиент и сервер следуют спецификации API, коммуникация будет происходить так, как ожидается

Программы Unix взаимодействуют друг с другом с помощью конвейеров, которые представляют собой простые механизмы передачи сообщений от одного приложения к другому. Чтобы проиллюстрировать этот процесс, вспомните цепочку команд, которую можно запустить с терминала машины на базе Unix (например, компьютера с Mac или Linux):

```
$ history | less
```

Команда `history` выводит список всех команд, которые вы выполняли из профиля Bash. Список может быть длинным, поэтому можно разбить вывод истории на страницы с помощью команды `less`. Чтобы передать данные от одной команды к другой, используйте символ вертикальной черты (`|`), который дает оболочке команду перехватить вывод `history` и передать его на вход `less`. Мы говорим, что этот тип канала глупый, потому что его единственная задача — передача сообщений от одного процесса к другому.

Как показано на рис. 1.5, веб-API обмениваются данными через HTTP. На транспортном уровне передачи данных ничего не известно о конкретном протоколе API, который мы используем, поэтому он представляет собой наш глупый канал, в то время как сам API содержит всю необходимую логику для обработки данных.

API не должны изменяться, при этом вы можете изменять внутреннюю логику работы сервиса при условии, что она соответствует документации API. Это означает, что покупатели, взаимодействующие с API, должны иметь возможность продолжить обращаться к API точно так же, как и раньше, и получать при этом те же ответы. Это приводит к еще одной важной концепции в микросервисной архитектуре: *заменяемости (replaceability)*[1]. Идея заключается в том, что вы должны иметь возможность полностью заменить код, который скрыт за эндпоинтом, но при этом эндпоинт останется доступным, а значит и взаимодействие между сервисами

[1] *Ньюман С.* Создание микросервисов. — С. 30.

не нарушится. Теперь, когда вы понимаете, что такое API и как эти интерфейсы помогают осуществлять интеграцию сервисов, рассмотрим наиболее важные проблемы, возникающие при создании микросервисов.

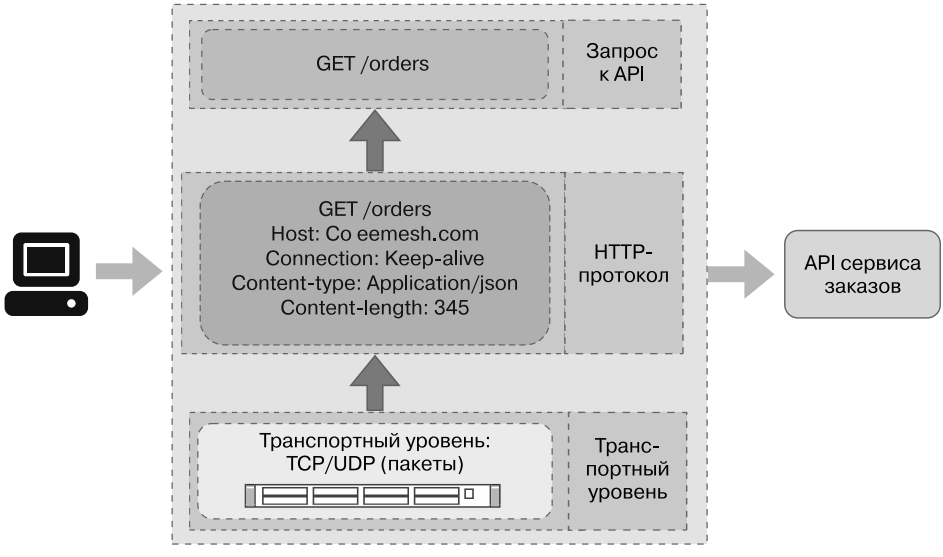

Рис. 1.5. Микросервисы взаимодействуют через API, используя транспортный уровень передачи данных, такой как HTTP-over-TCP

1.3. ПРОБЛЕМЫ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ

Как вы видели в подразделе 1.1.2, микросервисы дают существенные преимущества. Однако их использование также сопряжено со значительными трудностями. В этом разделе мы обсудим наиболее важные проблемы, которые я разделил на пять основных категорий, таких как:

- оптимальная декомпозиция сервиса;
- интеграционные тесты микросервисов;
- обработка недоступности сервиса;
- трассировка распределенных транзакций;
- повышение сложности эксплуатации и накладных расходов на инфраструктуру.

Все проблемы и трудности, которые мы обсуждаем в этом разделе, можно решить с помощью конкретных паттернов и стратегий — часть из них подробно описана в книге. Вы также найдете ссылки на другие ресурсы, где рассматриваются эти вопросы. Главное, чтобы вы понимали, что микросервисы не являются панацеей от всех проблем, с которыми сталкиваются монолитные приложения.

1.3.1. Оптимальная декомпозиция сервиса

Одна из самых важных задач при проектировании микросервисов — это декомпозиция системы. Мы должны разбить систему на слабо связанные, но достаточно независимые компоненты с четко заданными границами. Вы можете выявить нецелесообразную связь между сервисами, если всякий раз, когда вы меняете один сервис, приходится менять и другой. Это означает, что либо между сервисами неустоявшиеся правила обмена, либо оба компонента достаточно зависимы друг от друга, чтобы можно было объединить их в один. Неспособность разбить систему на независимые микросервисы может означать то, что Крис Ричардсон, автор книги «Микросервисы. Паттерны разработки и рефакторинга», называет *распределенным монолитом*, — решение, когда вы объединяете все проблемы монолитной и микросервисной архитектур и не получаете при этом преимуществ ни от одной из них. В главе 3 вы узнаете, какие паттерны проектирования и стратегии декомпозиции сервисов могут помочь разбить систему на микросервисы.

1.3.2. Интеграционные тесты микросервисов

В подразделе 1.1.2 мы говорили, что микросервисы обычно легче тестировать, и их наборы тестов выполняются быстрее. Однако интеграционные тесты микросервисов могут быть значительно сложнее, особенно в тех случаях, когда для выполнения одной транзакции требуется взаимодействие нескольких микросервисов. Когда все ваше приложение работает в рамках одного процесса, тестировать интеграцию между различными компонентами довольно легко, и для этого в основном требуются хорошо написанные модульные тесты. В контексте микросервисов для тестирования интеграции между несколькими сервисами необходимо иметь возможность запускать их все с параметрами, как в продакшен-окружении.

Для тестирования интеграции микросервисов можно использовать различные стратегии. В первую очередь нужно убедиться, что каждый сервис имеет хорошо документированный и правильно реализованный API. Вы можете проверить реализацию API на соответствие спецификации с помощью таких инструментов, как Dredd и Schemathesis, которые мы рассмотрим в главе 12. Вы также должны убедиться, что клиент использует API именно так, как предписано его документацией. Для этого вы можете написать модульные тесты для клиента, сгенерировав по документации API ответы от сервиса[1]. Но ни один из этих тестов не будет достаточным без полноценного сквозного тестирования (end-to-end, e2e-тестирование), при котором запускаются реальные микросервисы, отправляющие запросы друг другу.

---

[1] Чтобы больше узнать о процессе разработки API и о том, как использовать mock-серверы при разработке клиента, посмотрите мою презентацию API Development Workflows for Successful Integrations («Процессы разработки API для успешных интеграций») // Manning API Conference, 3 августа 2021 года. https://youtu.be/SUKqmEX_uwg.

1.3.3. Обработка недоступности сервиса

Необходимо убедиться, что приложения устойчивы к недоступности сервисов, тайм-аутам подключений и запросов, ошибочным запросам и т. д. Например, когда мы размещаем заказ через приложение доставки еды, такое как Uber Eats, Delivery Hero или Deliveroo, запускается цепочка запросов между сервисами для обработки и доставки заказа, и любой из этих запросов может завершиться неудачей. Рассмотрим процесс, который происходит, когда пользователь размещает заказ (см. рис. 1.6 для иллюстрации цепочки запросов).

1. Покупатель размещает заказ и оплачивает его. Заказ размещается с помощью сервиса заказов, а для обработки платежа тот взаимодействует с сервисом платежей.
2. Если оплата прошла успешно, сервис заказов делает запрос в сервис кухни для постановки в очередь заказа на приготовление.
3. После приготовления заказа сервис кухни делает запрос в сервис доставки, чтобы запланировать доставку.

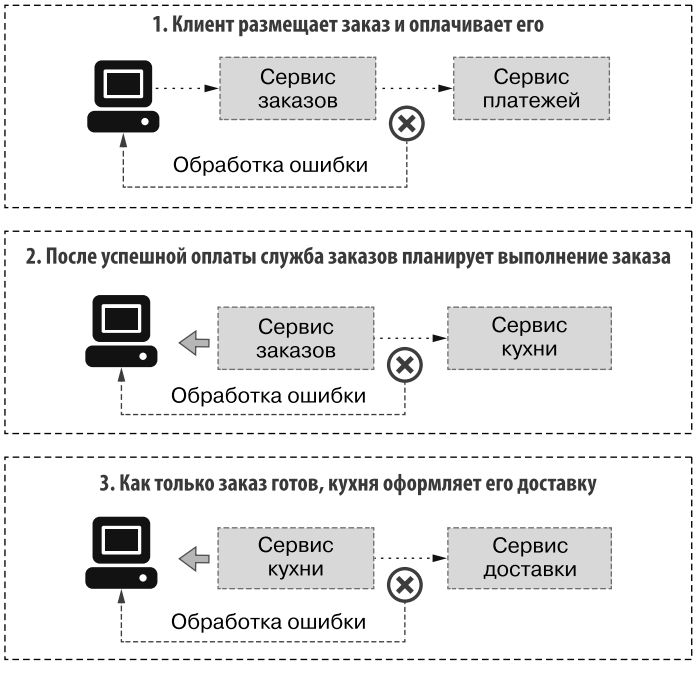

Рис. 1.6. Микросервисы должны быть устойчивы к таким событиям, как недоступность сервиса, тайм-аут запроса и ошибки обработки, полученные от других сервисов, и либо повторно выполнять запросы, либо выдавать пользователю понятный ответ