1 Общий обзор контрактного тестирования

В этой главе:

- ✓ О чем эта книга и почему эта тема важна
- ✓ Общий обзор контрактного тестирования
- ✓ Зачем нужно контрактное тестирование
- ✓ Когда стоит его использовать и когда оно бесполезно
- ✓ Общие принципы контрактного тестирования
- ✓ Контрактное тестирование в действии

Представьте следующий сценарий. Пятница, 16:00, вы решили развернуть небольшое обновление, в котором немного изменился ответ для одного из запросов API. Все автоматические тесты в вашем пайплайне прошли успешно, и теперь вы уверены, что ничего не сломали. Вы запускаете развертывание, и через несколько минут после его окончания другая команда сообщает вам, что на продакшене что-то сломалось. Конечно, изменение, которое вы только что развернули, не может быть причиной ошибки, просто потому что оно очень маленькое и все тесты прошли успешно... так ведь? Как выяснится потом,

у команды, которая использует ваш API, были некоторые ожидания, и лишь благодаря инциденту вы узнали, что они отличались от ваших. Но теперь, вместо того чтобы строить планы на выходные, вам придется разбираться с багом на продакшене.

Во время разбора инцидента выяснилось, что команда, обнаружившая проблему, написала интеграционные тесты API. Раньше после ваших изменений эти тесты не выполнялись, потому что они находятся в отдельном репозитории и запускаются в другом пайплайне. Кроме того, они печально известны как медленные и ненадежные. По результатам постмортема было решено найти альтернативные способы отслеживания подобных проблем, и кто-то предложил использовать контрактное тестирование. Вы уже слышали этот термин, но не знаете, как такое тестирование решит вашу проблему.

Что ж, вы правильно выбрали источник информации! Эта книга раскрывает все важные аспекты контрактного тестирования: что это, как оно работает, чем отличается от других типов тестирования и почему вам стоит им заниматься. Мы рассмотрим техническую реализацию, я покажу, как использовать актуальные инструменты и с нуля внедрять контрактное тестирование в процесс непрерывной интеграции. Моя основная цель — убедить вас, что контрактное тестирование является неотъемлемой частью вашей тестовой стратегии, особенно если вы работаете с архитектурой микросервисов.

Переходя от монолитных приложений к микросервисам, команды сталкиваются с новыми трудностями в тестировании, поскольку количество точек взаимодействия между двумя сервисами может увеличиваться экспоненциально.

Проверить все отдельные микросервисы вместе в выделенной тестовой среде тоже не простая задача. Команды могут развертывать изменения в разное или, наоборот, в одно и то же время, что может привести к неожиданным изменениям в системе, ломающим тестовую среду. Традиционные техники тестирования плохо масштабируются для сложных систем, поэтому командам важно разобраться, как с помощью контрактного тестирования убедиться, что вносимые изменения не оказывают негативного влияния на работу других команд.

1.1. Общий обзор контрактного тестирования

Согласно самому простому определению, *контрактное тестирование* проверяет, что две системы имеют одинаковые ожидания относительно друг друга. Например, если веб-приложение отправляет `GET`-запрос к сервису данных, то оно имеет ожидания, как сервис ему ответит. Контрактное тестирование отражает взаимодействие между двумя системами и сохраняет его в контракте, которого должны придерживаться обе стороны.

Общая схема похожа на подписание контракта в обычной жизни. Хороший пример — как я оформляла контракт с издательством Manning. В договоре подробно прописаны требования к Manning — опубликовать эту книгу, и в то же время указано, что́ издательство ожидает от меня. Если в соглашение нужно внести изменения, необходимо уведомить обоих участников. Изменения вносятся только в том случае, если их подтвердит каждая из сторон. В этом смысле контрактное тестирование работает аналогично. Если сервис неожиданно вернет информацию в формате, отличном от ожидаемого веб-приложением, то это несоответствие будет зафиксировано и проанализировано обеими системами до того, как изменения попадут на продакшен.

Чтобы лучше понять суть контрактного тестирования, полезно разобраться, как работает традиционное интеграционное тестирование API. Для примера возьмем веб-приложение, которое отображает данные пользователя, показанные на рис. 1.1.

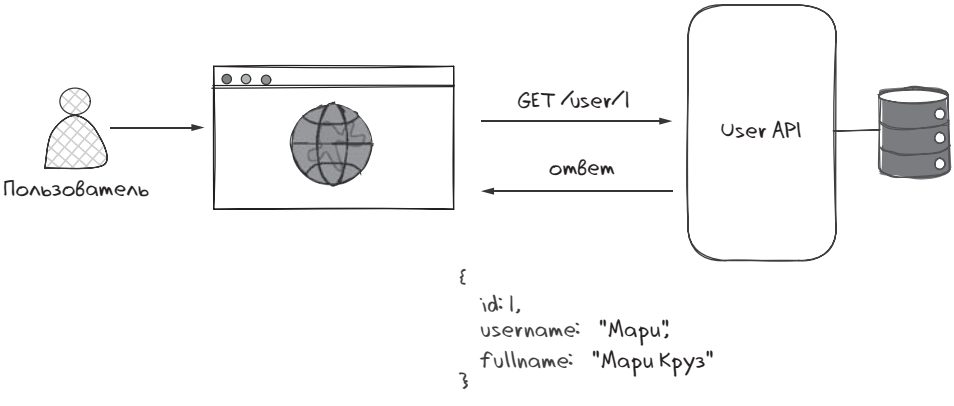

Рис. 1.1. Пример веб-приложения, которое взаимодействует с User API. Когда пользователь отправляет запрос на получение данных, API возвращает ожидаемый ответ

Простой тест API-интеграции, который показан на рис. 1.1, отправляет `GET`-запрос реальному сервису User API и проверяет, что тот вернул корректные данные: идентификатор, логин и полное имя пользователя.

Как и интеграционное, контрактное тестирование проверяет, что при отправке `GET`-запроса от веб-приложения к User API ожидания обеих систем выполняются. Главное отличие состоит в том, что веб-приложению и User API не нужно общаться напрямую. Они взаимодействуют через общий контракт, как показано на рис. 1.2.

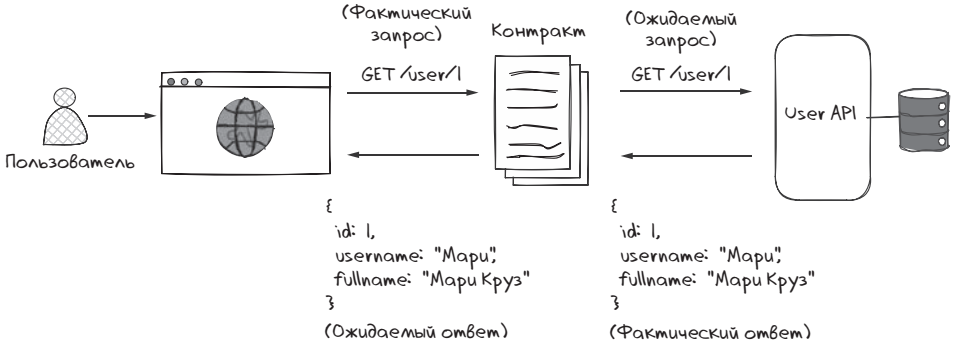

Рис. 1.2. Веб-приложение и User API взаимодействуют не напрямую, а через общий контракт

Контракт хранится в JSON-файле, который содержит имена двух взаимодействующих систем: в данном случае это веб-приложение и User API. Контракт также перечисляет все действия между ними (мы рассмотрим их позже в главе 3). Веб-приложение записывает в контракт фактический запрос и ответ, ожидаемый от User API. Далее контракт загружается на общий брокер, к которому User API может обратиться напрямую. API берет контракт из общего брокера и воспроизводит запрос, ожидаемый от веб-приложения. Затем специальное приложение для контрактного тестирования сопоставляет фактический ответ User API и ответ, который ожидает веб-приложение.

1.2. Почему важно применять контрактное тестирование

В идеале между двумя системами не должно возникать критических ошибок. Однако мы знаем, что, к сожалению, это все же случается. Например, представьте, что кто-то изменил название поля User API с `fullname` на `name` и развернул это на продакшене. В таком случае системы становятся несовместимыми, потому что веб-приложение ожидает поле `fullname`, как показано на рис. 1.3.

На данном этапе пользователи, которые обращаются к веб-приложению, не видят свое имя и начинают писать разработчикам жалобы. В худшем случае окружение продакшена будет сломано, пока команда, ответственная за User API, не откатит изменения или команда, ответственная за веб-приложение, не обновит свой код и не заменит название поля `fullname` на `name`.

Если обе команды используют контрактное тестирование, то эта несовместимость будет отловлена раньше, что позволит им обсудить изменение контракта. Если команда веб-приложения не согласится с изменением, то его придется откатить.

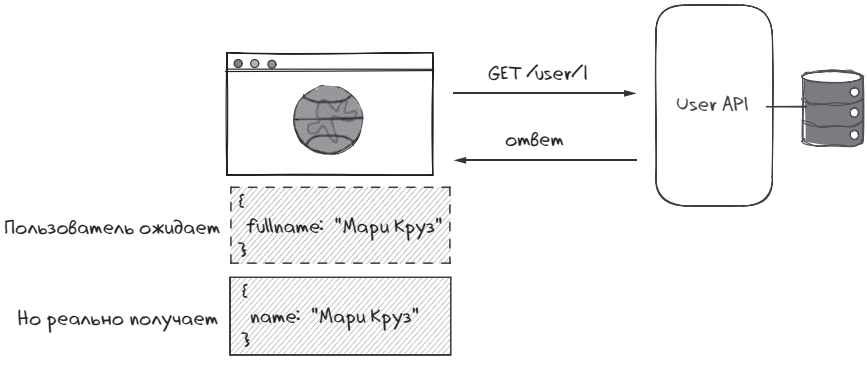

Рис. 1.3. Веб-приложение взаимодействует с User API, но их ожидания относительно названия поля, содержащего имя пользователя, не совпадают

Еще один плюс контрактного тестирования заключается в том, что с его помощью провайдеры данных могут вводить новые требования независимо и безопасно. Например, представим, что у команды API, кроме команды веб-приложения, есть еще один потребитель — пусть это будет мобильное приложение, — который хочет отображать дополнительную информацию о пользователе, например его любимые товары. Мобильное приложение может создать отдельный контракт, в котором будут описаны его ожидания от User API и которым сможет руководствоваться команда API, чтобы безопасно внедрять изменения совместно с командой мобильного приложения, которая не зависит от контракта с командой веб-приложения, как показано на рис. 1.4.

Если мобильное приложение отправит `GET`-запрос в эндпоинт `/user/1`, то оно получит дополнительное поле `favorites` (предпочтения), но это не создаст проблем для веб-приложения, потому что его контракт с User API остается в силе. При этом API все еще соответствует требованиям как мобильного, так и веб-приложения относительно дополнительного поля `favorites`.

Контракты, созданные мобильным и веб-приложением, также загружаются на общий брокер, что позволяет нескольким командам легко просматривать их. Кроме того, команде User API становится проще отслеживать, кто является их потребителем. Общий брокер для всех контрактов облегчает визуализацию зависимостей между командами, что позволяет быстрее обнаруживать изменения в контрактах.

Еще одна причина, по которой важно использовать контрактное тестирование, заключается в том, что если ошибка обнаружена поздно, то ее стоимость возрастает экспоненциально. Рафаэла Азеведо (Rafaela Azevedo) детально описала эту проблему в одном из своих постов (https://mng.bz/Dpa0). Контрактное тестирование сочетает в себе лучшие качества интеграционного и юнит-тестирования:

высокий уровень достоверности первого, скорость и низкие расходы на сопровождение и выполнение второго. Контрактное тестирование также может минимизировать количество интеграционных или сквозных (end-to-end, E2E) тестов, которые, как показано, являются медленными, хрупкими, а также подвержены ошибкам.

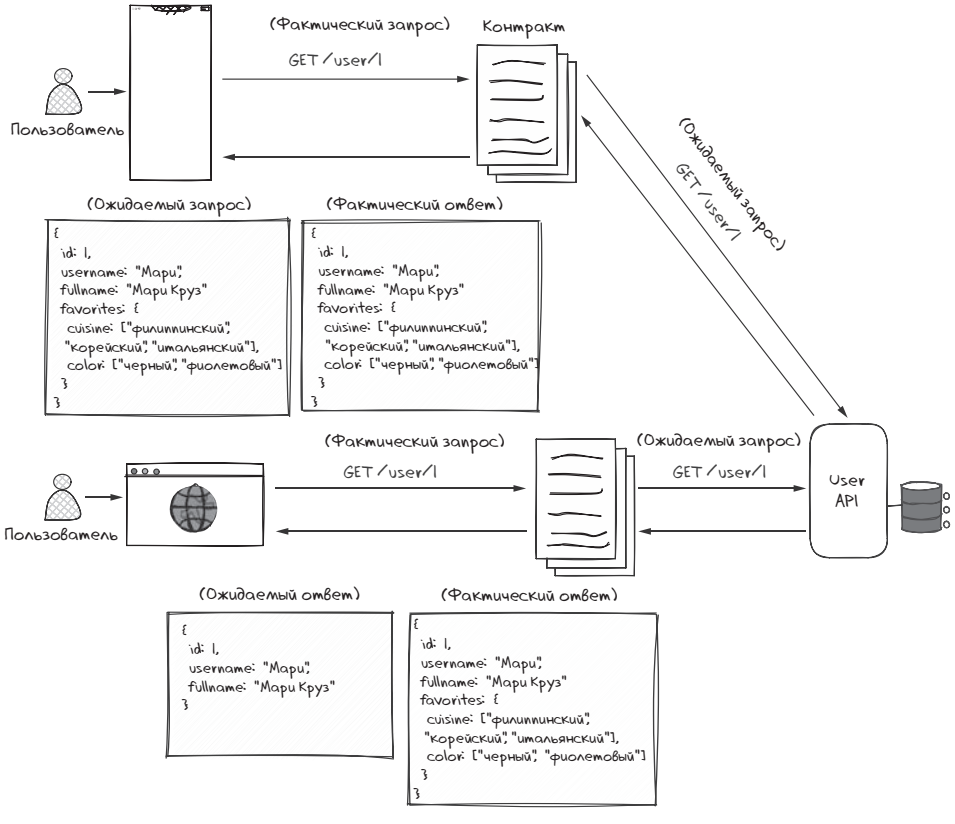

Рис. 1.4. User API с двумя потребителями и двумя независимыми контрактами

СОВЕТ Если у вашей команды уже есть большой набор интеграционных тестов, то многие из них можно перенести на более ранний этап разработки — уровень контрактов. Однако нельзя полностью отказаться от интеграционного или сквозного тестирования, потому что контрактное не выявляет проблем с окружением из-за ошибок конфигурации или непредвиденного поведения, вызванного побочными эффектами системы. Мы рассмотрим это подробнее в главе 2.

Помимо обнаружения ошибок, которые могут возникнуть даже при хорошем взаимодействии между командами, есть и другие причины использовать контрактное тестирование (и это далеко не полный список). Контрактное тестирование:

- ускоряет процесс разработки и предоставляет быструю обратную связь, поскольку контрактные тесты не обмениваются данными с другими системами;
- позволяет командам развертывать изменения независимо и безопасно;
- упрощает отладку, поскольку область проверки в контрактных тестах значительно меньше;
- позволяет разработчикам при необходимости запускать тесты и обновлять их локально, обеспечивая совместную ответственность за написание тестов;
- способствует улучшению качества кода, устраняя необходимость вводить требования, которые не будут использоваться;
- гарантирует, что до команд будут донесены любые критические изменения, поскольку контракт всегда будет их учитывать, особенно если вы используете специализированные инструменты, которые создают контракт из кодовой базы разработки.

1.3. Когда имеет смысл использовать контрактное тестирование?

Контрактное тестирование целесообразно использовать для любой точки интеграции между двумя системами, особенно если необходимо выявить проблемы на ранней стадии. Эта точка может быть между веб- (или мобильным) приложением и API или между двумя сервисами данных, взаимодействующими по разным протоколам, наиболее распространенным из которых является HTTP.

Вы можете использовать контрактное тестирование, если работаете только с двумя системами, которые взаимодействуют друг с другом, но лучше всего оно раскрывается при работе с микросервисной архитектурой, особенно при большом количестве микросервисов. Интеграционными тестами API и сквозными тестами трудно управлять и их сложно масштабировать. Контрактное тестирование — это отличная альтернатива.

СОВЕТ Контрактное тестирование можно применять для двух связанных систем. Однако если вы создадите контракт для веб-приложения, взаимодействующего с несколькими микросервисами, то будут применимы те же принципы.

В некоторых случаях нецелесообразно использовать контрактное тестирование. Вот несколько примеров.

- Тестирование всей бизнес-логики провайдера и побочных эффектов конкретной функциональности (это должно быть покрыто функциональным тестированием на стороне провайдера).
- Тестирование компонентов пользовательского интерфейса, таких как внешний вид или требования доступности.
- Тестирование других требований, таких как производительность или безопасность.
- Использование публичного API, если вы не знаете, кто потребляет ваши данные.

Однако в главе 11 мы рассмотрим, как использовать контрактное тестирование, если вы работаете со сторонним или публичным API с помощью подхода, который называется двунаправленным контрактным тестированием (bi-directional contract testing).

1.4. Общие принципы контрактного тестирования

Существует несколько подходов к применению контрактного тестирования. В последующих главах мы рассмотрим их подробно. А в рамках этой вступительной части я опишу только общие принципы, которые лежат в основе тестирования, ориентированного на потребителя (consumer-driven). В этом случае потребитель (подробнее о них см. главу 3, раздел 3.1) создает контракт и раздает его провайдерам данных.

Ранее в этой главе мы рассмотрели, как работают традиционные интеграционные тесты API. Я настоятельно рекомендую разобраться в этой теме, прежде чем переходить к контрактному тестированию. Для этого можно почитать о тестировании API и REST-ful API. Особенно я рекомендую книгу Марка Винтерингема Testing Web APIs[1]. Контрактное тестирование можно рассматривать как форму интеграционного, но с некоторыми особенностями, сближающими его с асинхронным юнит-тестированием.

Представьте, что вы купили дом в Великобритании. Когда кто-то в этой стране продает или покупает недвижимость, стороны обговаривают все детали сделки и хранят свои копии договора до самого конца процесса. Представьте, что эти стороны — распределенные команды разработки, которые реализуют бэкенд или фронтенд на разных языках программирования. Документация для API часто пишется в разных форматах: в виде страницы в вики, OpenAPI-спецификации

[1] Винтерингем М. «Тестирование веб-API». СПб.: Питер.

(www.openapis.org/) и др. Давайте обсудим OpenAPI-спецификацию и выясним, чем она отличается от контракта, созданного в процессе контрактного тестирования.

Спецификация OpenAPI, которая ранее называлась Swagger-спецификацией (https://swagger.io/docs/specification/about/), — это стандарт, который описывает детали API и делает их понятными для использующих их людей. Давайте рассмотрим пример такой спецификации:

```
openapi: 3.0.3
info:
  title: Contract Testing - OpenAPI 3.0
  version: 1.0.0
tags:
  - name: user
    description: Операции с пользователем
paths:
  /user/{id}:
    get:
      summary: Получить существующего пользователя
      operationId: getUser
      parameters:
        - name: id
          in: path
          description: вернуть Id пользователя
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: операция прошла успешно
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
components:
  schemas:
    User:
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 10
        firstName:
          type: string
          example: Мари
        lastName:
          type: string
          example: Круз
```

Спецификация OpenAPI описывает структуру API, определяет доступные эндпоинты и их свойства и может использоваться для написания интеграционных

тестов. Однако само по себе использование OpenAPI не предотвращает появление ошибок интеграции, поскольку команды могут развертывать несколько изменений в одной тестовой среде. Контракт, созданный в ходе тестирования, помогает избежать интеграционных багов, потому что информирует потребителей и провайдеров о наличии изменений при внесении их в пайплайн непрерывной интеграции (continuous integration, CI) еще до их развертывания в тестовой среде.

Интеграционные тесты также часто пишутся одной командой, например той, которая сопровождает или использует API. Такой подход может вызвать трудности при обмене ресурсами и знаниями между разными сервисами или приложениями. При этом в контрактном тестировании, ориентированном на потребителя, тесты пишутся командой, которая использует API (также поддерживаются сообщения, о которых мы поговорим подробнее в главе 8), а затем проверяются командой, которая его сопровождает.

1.4.1. Контрактное тестирование со стороны потребителя

В контрактном тестировании, ориентированном на потребителя, бóльшая часть тестов будет выполняться потребителем, поскольку контракт в этом случае задает именно он. Давайте рассмотрим пример. Представьте, что фронтенд-разработчик написал контрактный тест с помощью приложения для контрактного тестирования Pact (https://pact.io/), потому что хочет убедиться, что на странице пользователя не возникнет ошибки в данных, которые она должна отображать. Процесс написания контрактного теста со стороны потребителя показан на рис. 1.5.

Контрактный тест на стороне потребителя, изображенный на рис. 1.5, состоит из следующих шагов.

1. Фронтенд-разработчик, или в общем случае потребитель, задает правила взаимодействия с мок-провайдером (mock provider, имитация провайдера). Под *взаимодействием* (interaction) следует понимать запрос и ответ, которые будут зарегистрированы у мок-провайдера. *Запрос* (request) описывает, что потребитель будет отправлять провайдеру, а *ответ* (response) — что провайдер должен вернуть.
2. Фронтенд-разработчик во время юнит-тестирования запускает тест к мок-провайдеру, и этот провайдер проверяет, что запрос, отправленный потребителем, зарегистрирован в правилах взаимодействия. Если да, то мок-провайдер возвращает ожидаемый ответ, а потребитель подтверждает, что этот ответ ему понятен.
3. Если все тесты этого типа прошли успешно, то автоматически генерируется контракт, который включает заданное взаимодействие и затем загружается на

общий брокер, к которому имеет доступ настоящий провайдер данных. Существуют разные способы сохранения контракта, и мы рассмотрим их в главе 9.

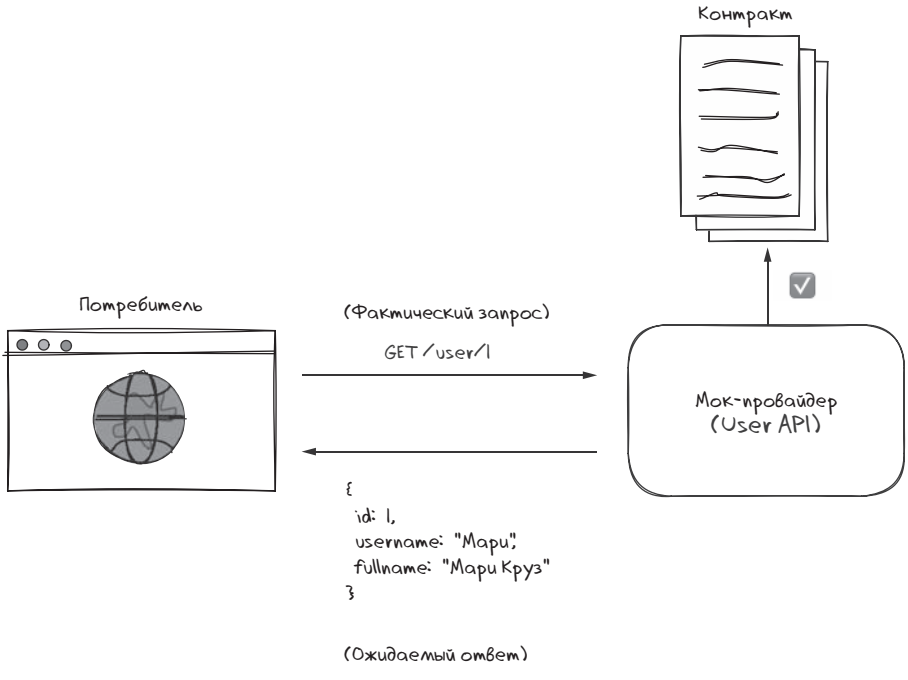

Рис. 1.5. Контрактный тест между потребителем и мок-провайдером данных

Я подробнее расскажу о написании контрактного теста, ориентированного на потребителя, в главе 4.

1.4.2. Контрактное тестирование со стороны провайдера

На стороне провайдера, чей код обычно находится в отдельном репозитории, один из бэкенд-разработчиков получает уведомление о том, что потребитель сгенерировал новый контракт. Провайдер проверяет его в рамках своего контрактного тестирования. Аналогично тесту потребителя, провайдер взаимодействует не с реальным потребителем, а с его имитацией (моком). Контрактный тест со стороны провайдера, изображенный на рис. 1.6, в общем случае будет состоять из следующих шагов.

1. Провайдер получает контракт, созданный потребителем, из общего брокера.
2. Используя приложение для контрактного тестирования, он выполняет ожидаемый запрос и отправляет свой ответ.