Тестирование веб-API

Марк Винтерингем

Вступительное слово
Джанет Грегори и Лизы Криспин

MANNING

Testing Web APIs

MARK WINTERINGHAM
FOREWORD BY JANET GREGORY AND LISA CRISPIN

MANNING
SHELTER ISLAND

Тестирование веб-API

МАРК ВИНТЕРИНГЕМ
ВСТУПИТЕЛЬНОЕ СЛОВО ДЖАНЕТ ГРЕГОРИ И ЛИЗЫ КРИСПИН

ПИТЕР®
Санкт-Петербург · Москва · Минск
2024

ББК 32.988.02-018
УДК 004.738.5
В50

Винтерингем Марк

В50 Тестирование веб-API. — СПб.: Питер, 2024. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2092-5

Веб-интерфейсы — лицо приложения, и они должны быть совершенными. Внедрение программы автоматизированного тестирования — лучший способ убедиться, что ваши API готовы к работе.

«Тестирование веб-API» — это уникальное практическое руководство, включающее в себя описание всех этапов: от начального проектирования набора тестов до методов документирования, реализации и предоставления высококачественных API. Вы познакомитесь с обширным набором методов тестирования — от исследовательского до тестирования продакшен-кода, а также узнаете, как сэкономить время за счет автоматизации с использованием стандартных инструментов. Книга поможет избежать многих трудностей при тестировании API.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1617299537 англ.

ISBN 978-5-4461-2092-5

Краткое содержание

ЧАСТЬ 3
РАСШИРЯЕМ НАШУ СТРАТЕГИЮ ТЕСТИРОВАНИЯ

Оглавление

Для Стеф:
Я обещаю, что закончу ремонт на кухне.

Вступительное слово

«Тестирование веб-API» содержит значительно больше информации, чем можно ожидать от книги с таким названием. Тестирование API представлено как часть единой стратегии тестирования, учитывающей риски. Марк знакомит вас с полезными визуальными моделями, задает вопросы, чтобы заставить думать, и ведет до последних страниц в качестве участника, а не пассажира.

Прежде чем углубиться в детали, в книге предлагается глава о том, зачем мы вообще тестируем, а также как идентифицировать различные типы рисков. Марк делает еще шаг вперед в исследовании этой важной темы — сопоставляет риски с качественными характеристиками и связывает их со стратегией тестирования.

Автор указывает предварительные условия для получения максимальной отдачи от книги. Практикующие читатели, знакомые с написанием кода, HTTP, инструментами разработки и тестирования, изучат новые инструменты и методы, чтобы понять все аспекты поведения их API. Тем не менее книга полезна и людям, которые не имеют всех этих навыков, поскольку чтение даст им общее понимание проблем и вдохновит на выполнение учебных упражнений.

Многие примеры в книге, а также упражнения основаны на реальном проекте с использованием работающего приложения. Вы будете изучать продукт, его область работы, историю и ошибки — все как в реальной жизни. Пользовательский интерфейс продукта поможет ознакомиться с его функциями.

Нам нравится, что эта книга помогает людям применять целостный подход к тестированию API. Проведите беседы о качестве, чтобы договориться о его желаемом уровне, и разработайте стратегию для его достижения, сотрудничая с заинтересованными сторонами. Автор пишет об этом в главе 4:

> *Хорошая стратегия тестирования характеризуется целостностью. Она должна фокусироваться на разных аспектах — везде, где велики риски снижения качества продукта.*

Также нас привлекает то, что Марк одинаково ценит исследовательское тестирование и автоматизацию как элементы стратегии тестирования API. Он очень прагматичен в обоих случаях, перечисляя плюсы и минусы и иллюстрируя их конкретными примерами. При этом он признает, что каждая команда имеет свой собственный контекст. Еще одно качество книги, которое нам нравится, заключается в том, что читателю постоянно напоминают о необходимости использовать физические или виртуальные средства визуализации всего, что обсуждается.

Книга проведет вас от модели стратегии к ее планированию и реализации в соответствии с вашим контекстом. Последние пять глав посвящены расширенной автоматизации тестирования API. В них вы глубоко изучите тестирование контрактов, производительности, безопасности, а также тестирование в рабочей среде. Вы можете самостоятельно выбрать очередность изучения этих тем.

Одна из наших любимых моделей, широко используемая в книге, основана на идее Джеймса Линдсея (James Lyndsay). Это диаграмма Венна, демонстрирующая различие наших представлений о продукте и его реализации. Она помогает нам задавать такие вопросы, как «Кто будет использовать этот ответ API?» или «Что, если я нажму эту кнопку тысячу раз?». Это один из многих способов, помогающих нам мыслить нестандартно.

Использование API продолжает расти по мере того, как все больше приложений переходят на микросервисы и облачные решения. Методы и модели, описанные в этой книге, позволяют создавать высококачественные и надежные API. Эти же модели и методы могут быть адаптированы для многих других видов тестирования. Прочтите эту книгу, и вы разовьете свои навыки тестирования.

— *Джанет Грегори* (Janet Gregory), консультант, автор, спикер в Dragonfire Inc.; соучредитель сообщества The Agile Testing Fellowship

— *Лиза Криспин* (Lisa Crispin), консультант по тестированию, автор и соучредитель сообщества The Agile Testing Fellowship

Предисловие

Я всегда чувствовал, что когда впервые начал тестировать API, то уже опоздал. Программное обеспечение как услуга (software as a service) получило широкое распространение, и микросервисы набирали популярность. Я видел, как разработчики в моей команде успешно тестировали API в контексте автоматизации, но только когда мне посчастливилось поработать с одним из них, мой путь к тестированию API начался по-настоящему. Спасибо, Упеш!

Однако по мере того как я развивал свои навыки и начал делиться знаниями с помощью видеокурсов и очного обучения, стало очевидно, что многие еще не начали свой путь или уже начали, но хотят узнать больше. Это стало мотивацией для написания книги: я хотел дать читателям широкое представление о множестве способов, которыми мы можем тестировать веб-API, и рассказать, как они работают.

Когда я впервые стал обучать других тестированию API, то сосредоточился на помощи в понимании и использовании возможностей HTTP, чтобы работать быстрее и качественнее. Но по мере подготовки материала книги я осознал, что нужно охватить значительно более широкий круг вопросов. Вот почему в этой книге мы рассмотрим методы тестирования, которыми можем воспользоваться на всех стадиях разработки веб-API, от постановки задач и написания первой строки кода до создания сложной автоматизации, которая дает ценную обратную связь.

Я надеюсь, что, изучив материал книги, вы получите в свое распоряжение инструменты, которые помогут стать лучше в тестировании API, независимо от вашего опыта и роли в команде.

Благодарности

Прежде всего я хотел бы поблагодарить тех, кто активно помогал мне в создании этой книги: моих редакторов — Кристину Тейлор (Christina Taylor), которая была терпелива, когда я взял перерыв, чтобы снова стать отцом, и Сару Миллер (Sarah Miller), которая помогла мне закончить эту книгу, а также всему производственному персоналу издательства Manning. Я также хотел бы поблагодарить Эбби Бэнгсер (Abby Bangser) и Билла Мэтьюза (Bill Matthews), которые нашли время поговорить о тестировании в производстве и тестировании безопасности соответственно. Также спасибо моему коллеге по автоматизации в тестировании Ричарду Брэдшоу (Richard Bradshaw), с которым мы много дискутировали о тестируемости и стратегии, что помогло обогатить главу с описанием стратегий тестирования. И мы еще долго будем обсуждать отношение к автоматизации тестирования. Наконец, спасибо всем, от кого я получил обратную связь: Alberto Almagro, Allen Gooch, Amit Sharad Basnak, Andres Sacco, Andy Kirsch, Andy Wiesendanger, Anne-Laure Gaillard, Anupam Patil, Barnaby Norman, Christopher Kardell, Daniel Cortés, Daniel Hunt, Ernesto Bossi, Ethien Daniel Salinas Domínguez, Hawley Waldman, Henrik Jepsen, Hugo Figueiredo, James Liu, Jaswanth Manigundan, Jeffrey M. Smith, Jonathan Lane, Jonathan Terry, Jorge Ezequiel Bo, Ken Schwartz, Kevin Orr, Mariyah Haris, Mark Collin, Marleny Nunez Alba, Mikael Dautrey, Dr. Michael Piscatello, Brian Cole, Narayanan Jayaratchagan, NaveenKumar Namachivayam, George Onofrei, Peter Sellars, Prashanth Palakollu, Rajinder Yadav, Raúl Nicolás, Rohinton Kazak, Roman Zhuzha, Ronald Borman, Samer Falik, Santosh Shanbhag, Shashank Polasa Venkata, Suman Bala, Thomas Forys, Tiziano Bezzi, Vicker Leung, Vladimir Pasman, Werner Nindl, William Ryan, Yvon Vieville, and Zoheb Ainapore. Это было трудное, но важное чтение.

(Альберто Альмагро, Аллен Гуч, Амит Шарад Баснак, Андрес Сакко, Энди Кирщ, Энди Вьесенданжер, Энн-Лаура Гаиллард, Анупам Патил, Барнаби Норман, Кристофер Карделл, Дэниел Кортéс, Дэниел Хант, Эрнесто Босси, Этьен Дэниел Салинас Домингез, Холи Волдман, Хенрик Йепсен, Хуго Фигуйредо, Джеймс Лью, Джасвант Манигундан, Джеффри М. Смит, Джонатан Лейн, Джонатан

Терри, Йорге Езекуэл Бо, Кен Щварц, Кевин Орр, Мария Харис, Марк Коллин, Марлени Нунез Альба, Микаел Дотри, Майкл Пискателло, Брайан Коль, Нараянан Джаярачаган, НавинКумар Намачиваям, Джордж Онофрей, Питер Селларс, Прашантх Палаколлу, Райиндер Ядав, Рауль Никола́с, Рохинтон Казак, Роман Жужа, Рональд Борман, Самер Фалик, Сантош Шанбхаг, Шашанк Поласа Венката, Суман Бала, Томас Форис, Тициано Бецци, Викер Леунг, Владимир Пасман, Вернер Киндл, Вильям Рян, Ивон Вьевилль и Зохеб Айнапор)

Я также в долгу перед Лизой Криспин (Lisa Crispin) и Джанет Грегори (Janet Gregory) за их добрые слова и время, потраченное на написание предисловия к этой книге. Спасибо Джеймсу Линдсею (James Lyndsay), Робу Мини (Rob Meaney) и Эшу Винтеру (Ash Winter), чья работа помогла мне лучше понять ключевые аспекты тестирования и позволила поделиться этими знаниями в книге.

Есть люди, которые, сами того не понимая, помогли мне с этой книгой. Например, Упеш Амин (Upesh Amin) много лет назад любезно нашел время, чтобы однажды после обеда научить меня работать с HTTP, а Алан Ричардсон (Alan Richardson) навел на мысль о работе над книгой во время курса Marketing 101!

Эта книга является кульминацией моего опыта работы в сообществе тестировщиков, поэтому спасибо всем в Ministry of Testing[1] и многочисленным друзьям, которых я приобрел за эти годы на различных мероприятиях, устраиваемых сообществом тестировщиков. Благодарю всех, кто саркастически спрашивал: «О, ты пишешь книгу?» — я ценю бесплатную рекламу и мотивацию.

Но больше всего я хочу поблагодарить Стеф (Steph), которая всегда поддерживала меня во всех сумасшедших проектах, которыми я занимался, и терпеливо и вежливо поздравляла меня каждый вечер, когда я взволнованно говорил ей, что написал «еще три страницы!». И это продолжалось в течение целого года.

[1] Ministry of Testing — Министерство тестирования, также называемое MoT, представляет собой глобальное сообщество по тестированию программного обеспечения. — *Примеч. пер.*

О книге

Замысел этой книги преследует две цели. Первая цель — познакомить читателей с широким спектром различных техник в области тестирования, которые применимы к веб-API. По мере изучения материала вы узнаете, как выполнять различные виды тестирования, как оценить типы рисков и получаемую информацию. Вторая цель — помочь вам создать и внедрить стратегию тестирования, которая успешно сочетает в себе различные техники и эффективна в вашем конкретном случае.

ДЛЯ КОГО ЭТА КНИГА

При написании книги я попытался организовать материал так, чтобы помочь вам построить стратегию тестирования шаг за шагом. Однако в духе принципа применения разных стратегий для различных условий предлагается несколько способов, которыми вы можете воспользоваться, чтобы добиться успеха в тестировании.

Независимо от вашей мотивации, я настоятельно рекомендую прочитать часть 1 полностью. Глава 2 поможет освоиться в песочнице, которая используется в практических примерах. Она понадобится, если вы захотите опробовать различные методы, описанные в книге. Глава 3 обязательна к прочтению, потому что в ней подробно рассматриваются качество и риски, а также то, как они влияют на продукт. Я твердо убежден, что для успешного тестирования необходимо четко понимать, какую проблему вы пытаетесь решить. Если вы не знаете, в чем проблема, как вы можете быть уверены, что выбрали правильный подход, и как сможете оценить результат?

Остальной материал книги вы можете выбирать для чтения, исходя из своих предпочтений. Я надеюсь, что для некоторых книга станет руководством для поэтапного построения стратегии тестирования, а для других — удобным или

справочным пособием, информирующим о конкретных методах, ресурсах и навыках.

Новичкам в построении стратегии тестирования API

Книга построена таким образом, чтобы провести вас по всему пути с нуля до создания, реализации и выполнения успешной стратегии тестирования. Поэтому если вы новичок в тестировании API, последовательно изучайте главы, чтобы улучшать свои знания и навыки.

Улучшение существующей стратегии тестирования API

Не все читатели начинают с нуля. Вы можете быть членом команды, стремящейся улучшить существующую стратегию тестирования. В таком случае я рекомендую разобраться в построении предлагаемой стратегии и сравнить ее с вашей, чтобы определить возможные пробелы и недостатки последней. Этот анализ поможет определить действия по улучшению тестирования в вашей команде.

Реализация определенных методов тестирования

Некоторые читатели хотят узнать больше о конкретных техниках, и им не обязательно думать о более широкой картине (например, вам может быть поручено провести определенные тесты). Если ваша мотивация такова, я рекомендую сосредоточиться на интересующих вас темах, примерах и заданиях по тестированию. Другим проще понять, какое место занимает определенная техника в стратегии, попробовав ее, а затем уже расширять свое видение.

Структура книги

Двенадцать глав книги разделены на три части.

Часть 1. Разработка стратегии тестирования

В главе 1 мы спросим себя, зачем нужно тестирование и почему понимание его ценности помогает создать стратегию тестирования. Глава 2 знакомит с проектом песочницы, который мы будем использовать в практических примерах на протяжении всей книги, чтобы показать ряд методов, которые помогут быстро понять, что и для кого мы тестируем. Глава 3 объясняет, как определить цели,

которых мы стремимся достичь с помощью стратегии тестирования, и что помогает нам расставить приоритеты при выборе типа тестирования.

Часть 2. Введение тестов в нашу стратегию

В части 2, главах с 4-й по 7-ю, мы начинаем изучать техники тестирования веб-API. Я расположил главы в этой части книги так, чтобы они следовали жизненному циклу разработки программного обеспечения, начиная с идеи и заканчивая реализацией и сопровождением.

Эта часть книги завершается главой о том, как рабочий контекст влияет на нашу стратегию и как мы можем реализовать ее таким образом, чтобы поддержать работу команды. Затем мы используем эти знания, чтобы собрать воедино мероприятия по тестированию, сформировав основу нашей стратегии тестирования.

Часть 3. Расширение стратегии тестирования

В заключительной части книги, главах с 8-й по 12-ю, мы узнаем больше о разных техниках тестирования, а также расширим некоторые методы, о которых уже узнали. Важно отметить, что техники, которые мы рассмотрим в этом разделе, не обязательно являются более сложными или требуют большего мастерства. Однако они, возможно, потребуют больших затрат времени и более зрелой культуры тестирования, чтобы интегрировать их в стратегию.

Предварительные требования

Предполагается, что вы подходите к изучению этой книги, обладая рядом навыков и знаний.

HTTP

Чтобы протестировать веб-API, нам потребуется использовать HTTP. Мы подробно изучим применение этого протокола в различных техниках тестирования, однако эта книга не содержит введения в HTTP. Поэтому предполагается, что вы знакомы с основами HTTP, такими как:

- унифицированные идентификаторы/локаторы ресурсов;
- HTTP-методы;

- HTTP-заголовки;
- коды состояния;
- запросы и ответы.

Java

Для разделов, затрагивающих программирование, я решил использовать язык Java из-за его широкого распространения в мире разработки API. Это означает, что нам придется иметь дело с дополнительными шаблонами, поставляемыми с Java, но примеры, которые мы рассмотрим, будут понятны максимально широкой аудитории. Помимо этого, примеры содержат множество шаблонов проектирования для кода автоматизации, а подходы универсальны для разных языков. Поэтому я призываю вас прочитать примеры с кодом или даже попробовать их выполнить на практике. Однако для выполнения этих упражнений вы должны иметь практические знания в следующих областях:

- библиотеки;
- пакеты;
- классы;
- методы тестирования;
- утверждения.

Прочие инструменты

В этой книге рассматривается ряд инструментов, которые используются для поддержки различных мероприятий по тестированию:

- *DevTools* — расширение для большинства браузеров, помогающее отлаживать веб-страницы (https://developer.chrome.com/docs/devtools)
- *Postman* — инструментальная платформа, помогающая создавать и тестировать веб-API (https://www.postman.com)
- *Wireshark* — инструмент для перехвата HTTP-трафика, позволяющий перехватывать HTTP-трафик между API (https://www.wireshark.org)
- *Swagger* — инструмент проектирования, позволяющий создавать «живую» документацию, с которой вы можете взаимодействовать, чтобы узнать больше о веб-API (https://swagger.io)

- *WireMock* — инструмент для имитации веб-API, чтобы повысить управляемость тестирования (https://wiremock.org)
- *Pact* — инструмент тестирования контрактов, который проверяет интеграцию между веб-API (https://pact.io)
- *Apache JMeter* — инструмент для тестирования производительности и функциональности веб-API (https://jmeter.apache.org)

Если будет возможность, ознакомьтесь с этими инструментами, прежде чем начать путешествие по страницам книги.

О коде в книге

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и в обычном тексте. В обоих случаях исходный код отформатирован шрифтом фиксированной ширины, чтобы отделить его от обычного текста.

Комментарии в исходном коде часто удалялись из листингов, если код объяснялся в тексте. Аннотации кода выделяют важные концепции.

Вы можете получить фрагменты кода в liveBook-версии этой книги по адресу https://livebook.manning.com/book/testing-web-apis. Полный код примеров из книги доступен для загрузки с веб-сайта издательства Manning по адресу https://www.manning.com/books/testing-web-apis.

Кроме того, код есть в двух вспомогательных репозиториях, которые будут неоднократно упоминаться в тексте.

Restful-booker-platform

Restful-booker-platform — это наша платформа API-песочницы, на которой мы будем практиковаться в тестировании. Кодовую базу платформы можно найти по адресу https://github.com/mwinteringham/restful-booker-platform, а сведения об установке приведены в приложении к книге.

Источники, используемые в книге

Многие главы содержат заметки, примеры кода и сценарии тестирования производительности, которые можно просмотреть в соответствующих проектах

в следующем репозитории: https://github.com/mwinteringham/api-strategy-book-resources. Все разделы кода можно запускать локально.

ФОРУМ LIVEBOOK

Приобретая книгу «Тестирование веб-API», вы получаете бесплатный доступ к веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу https://livebook.manning.com/book/testing-web-apis/discussion. Информацию о форумах Manning и правилах поведения на них см. на https://livebook.manning.com/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

Об авторе

Марк Винтерингем — тестировщик, системный программист и COO (главный операционный директор) Ministry of Testing с более чем десятилетним опытом экспертизы в области тестирования. Участвовал в технологических проектах в разных областях, которые были отмечены наградами BBC, Barclays, правительства Великобритании и Thomson Reuters. Сторонник современных методов тестирования, основанных на оценке рисков. Обучает команды методам автоматизации тестирования, разработки через тестирование и исследовательского тестирования. Соучредитель Ministry of Testing, сообщества, занимающегося вопросами образования в области тестирования.

Иллюстрация на обложке

Изображение на обложке — «Boucbar de Siberie» (сибирский пастух), оно взято из коллекции Жака Грассе де Сен-Совера, опубликованной в 1788 году. Каждая иллюстрация была тщательно нарисована и раскрашена вручную.

В то время было легко определить по одежде, где живут люди и какова их профессия или положение в обществе. Издательство Manning отдает дань изобретательности и инициативам в области компьютерных технологий, создавая обложки, которые напоминают о культурном разнообразии многовековой давности и возвращают к жизни рисунки из старых изданий.

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.

Часть 1

Значение тестирования веб-API

В чем смысл тестирования? Вопрос может показаться саркастическим, но понимание целей и пользы поможет нам определить подходы к тестированию. Если мы решим, что тестирование — это просто нажатие кнопок и поломка разных вещей, то получим соответствующий результат. В действительности тестирование — это совокупность навыков, знаний и действий, которые помогают команде повысить качество своей работы. Вот почему, прежде чем мы перейдем к рассмотрению конкретных примеров, нам нужно понять важность правильного тестирования и то, как применять его для определенных задач в нужное время.

В части 1 мы рассмотрим, что и как тестировать, чтобы приносить реальную пользу как коллегам, так и клиентам. В главе 1 обсудим проблемы, связанные с качеством веб-API, и способы решать эти проблемы с помощью тестирования. В главе 2 наметим маршрут нашего путешествия по миру тестирования API. Наконец, в главе 3 будут подробно обсуждаться две ключевые концепции, которыми мы руководствуемся при тестировании: качество и риски.

1 Для чего и как мы тестируем веб-API

В этой главе

- ✓ Проблемы создания сложных API-платформ
- ✓ Значение и цель тестирования
- ✓ Как выглядит стратегия тестирования API и чем она может быть полезна.

Как мы можем гарантировать качество и ценность для конечных пользователей того, что создаем? Проблема, с которой мы сталкиваемся при выпуске высококачественного продукта, заключается в огромном количестве сложных действий и операций при его создании. Если мы хотим добиться высокого качества, необходимо преодолеть все сложности, а также разобраться, как работают наши системы и что пользователям требуется от них. Вот почему нам нужна стратегия тестирования, которая поможет лучше понять, что мы на самом деле создаем. Итак, прежде чем мы начнем наше путешествие по миру тестирования API, давайте сначала подумаем, почему программное обеспечение такое сложное и в чем польза тестирования.

1.1. ЧТО ПРОИСХОДИТ В ВАШИХ ВЕБ-API

В 2013 году правительство Великобритании разработало стратегию по переводу налоговой и таможенной службы (HMRC) на цифровой стандарт обслуживания. Цель стратегии состояла в том, чтобы перевести работу этих служб в онлайн-режим, улучшить качество услуг и сократить расходы.

К 2017 году налоговая платформа HMRC насчитывала более 100 цифровых услуг, созданных 60 группами в пяти различных центрах. Каждая из этих цифровых услуг поддерживается платформой взаимосвязанных веб-API, число которых постоянно растет. Количество API, созданных для поддержки этих сервисов, просто поражает. Когда я присоединился к проекту в 2015 году, услуг, команд и центров было примерно вдвое меньше, чем сейчас, но уже тогда платформа объединяла более 100 веб-API. С тех пор это число, несомненно, увеличилось. Возникает вопрос: как проект такого масштаба и сложности может предоставлять конечным пользователям высококачественные услуги?

Я привел этот пример, потому что он помогает выделить два уровня сложности при создании веб-API:

- Собственная сложность конкретного веб-API.
- Сложность, обусловленная взаимодействием нескольких веб-API, работающих на одной платформе.

Разобравшись с обеими, мы начнем понимать, зачем нужно тестирование и чем оно может нам помочь.

1.1.1. Собственная сложность веб-API

Может показаться немного банальным начать с вопроса: что такое веб-API? Но если мы потратим время на разбор структуры веб-API, то сможем узнать не только о том, что такое веб-API, но и в чем заключается его сложность. Рассмотрим, к примеру, представленную на рис. 1.1 визуализацию веб-API сервиса бронирования, который мы будем тестировать позже в этой книге.

На рисунке мы видим, что веб-API получает от клиентов данные о резервировании в форме HTTP-запросов, которые обрабатываются в API на разных уровнях. После завершения выполнения и сохранения резервирования веб-API отвечает через HTTP. Более детальное рассмотрение позволит понять, сколько всего происходит в рамках одного веб-API.

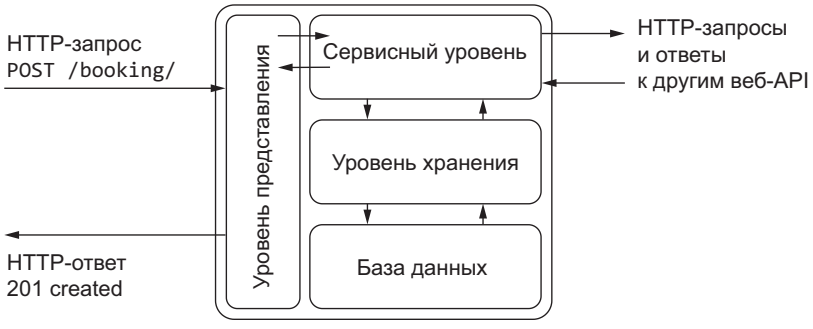

Рис. 1.1. Модель веб-API, изображающая его компоненты и то, как он работает

Прежде всего, на уровне представления HTTP-запрос на бронирование переводится в контент, который может быть прочитан другими уровнями. Сервисный уровень применяет к информации о бронировании бизнес-логику, например проверяет, действительно ли это бронирование и не конфликтует ли оно с другими бронированиями. Наконец, если обработанное бронирование необходимо сохранить, оно подготавливается на уровне хранения, а затем сохраняется в базе данных. Если все выполнено успешно, каждый уровень должен сообщить об этом, чтобы создать ответ, который веб-API отправит создателю запроса.

Каждый из рассмотренных слоев может быть построен по-разному, в зависимости от наших требований и вкусов. Например, можно разрабатывать веб-API с использованием архитектур REST, GraphQL или SOAP, каждая из которых имеет свои собственные шаблоны и правила.

Работа с REST

В этой книге мы будем работать преимущественно с веб-API, созданными с использованием архитектуры REST. Существует множество различных архитектурных стилей, но сегодня наиболее широко используется REST. Однако стоит отметить, что хотя GraphQL и SOAP имеют свои отличия, подходы к тестированию, которые мы рассмотрим, в равной степени применимы к этим типам архитектуры. На протяжении всей книги мы будем кратко отмечать, каким образом изучаемое может быть применено к любому архитектурному стилю.

Сервисный уровень содержит бизнес-логику, которая, в зависимости от контекста, может иметь множество настраиваемых правил. Аналогичный принцип

применяется к уровням хранения. Каждый из этих уровней опирается на зависимости, которые имеют свои собственные циклы разработки. Нам необходимо знать огромное количество информации, чтобы обеспечить высокое качество работы API.

Понимание того, что происходит в наших веб-API и как они помогают другим, требует времени и опыта. Да, можно достичь некоторого понимания, тестируя части по отдельности (к чему я всегда призываю команды; ознакомьтесь с выступлением Дж. Б. Райнсбергера (J. B. Rainsberger) «*Integrated Tests Are a Scam*», чтобы узнать больше: https://youtu.be/VDfX44fZoMc). Но это дает нам информацию только о части головоломки, а не обо всей задаче целиком.

1.1.2. Сложность взаимодействия многих веб-API

В случае с платформой HMRC с ее более чем 100 веб-API требуется понимание не только работы каждого из них, но и их взаимодействия друг с другом. Такие подходы, как микросервисная архитектура, помогают уменьшить сложность отдельных веб-API, делая их меньше и более конкретными. С другой стороны, при этом количество веб-API на платформе еще больше увеличивается. Как в этих условиях обеспечить актуальность наших знаний о платформе? И как следить за взаимодействиями каждого API с другими и контролировать, чтобы их соединения работали в пределах ожидаемого?

Чтобы создать высококачественный продукт, мы должны делать осознанный выбор. Это означает, что наши знания о работе веб-API, а также об их взаимодействиях как друг с другом, так и с конечными пользователями жизненно важны. Иначе мы рискуем столкнуться с проблемами из-за неверной интерпретации того, как работают наши системы. Именно с этой точки зрения тестирование важно для упорядочивания и сохранения понимания работы продуктов.

1.2. КАК ТЕСТИРОВАНИЕ ПОМОГАЕТ НАМ

Если мы как команда собираемся добиться успеха в тестировании, нам необходимо общее понимание его цели и значения. К сожалению, существует множество неправильных представлений о том, что такое тестирование. Чтобы помочь нам всем прийти к единому мнению, позвольте представить модель тестирования (рис. 1.2).

Рис. 1.2. Модель, которая помогает описать значение и цель тестирования

Эта модель, основанная на образе, созданном Джеймсом Линдсеем в его статье *«Why Exploration has a Place in any Strategy»* (http://mng.bz/o2vd), состоит из двух кругов. Левый круг соответствует нашим представлениям о продукте (воображению), а правый круг — реализации, то есть тому, что мы имеем в реальности. Цель тестирования — узнать как можно больше об этих кругах путем проведения тестов. Чем больше мы тестируем, тем больше продвигаемся в решении следующих задач:

- обнаруживаем потенциальные проблемы, которые могут повлиять на качество;
- разбираемся в том, что создаем, и приобретаем уверенность, что это именно тот продукт или услуга, которые мы хотим создать.

Чтобы лучше разобраться в этом, давайте рассмотрим пример, где команда предоставляет гипотетическую функцию поиска, качество которой мы должны обеспечить.

1.2.1. Представление

Круг представления отражает то, что мы хотим от нашего продукта, включая явные и неявные ожидания. В этом круге наше тестирование сосредоточено на том, чтобы узнать как можно больше об этих ожиданиях. Мы узнаем не только то, что было явно заявлено в письменной или устной форме, но также углубляемся в детали и устраняем двусмысленности в терминах и идеях.

В качестве примера допустим, что представитель бизнеса или пользователей (владелец продукта) сформулировал для своей команды требование: «Результаты поиска должны быть упорядочены по релевантности». Представленная явно информация говорит о том, что требуются результаты поиска, упорядоченные по релевантности. Однако мы можем раскрыть много неявной информации, проверяя идеи и концепции, лежащие в основе того, о чем нас просят. Чтобы разобраться в проблеме, можно задать следующие вопросы:

- Что подразумевается под *релевантными результатами*?
- Для кого результаты должны быть релевантны?
- Какая информация предоставляется?
- Как ее упорядочить по релевантности?
- Какие данные мы должны использовать?

Ответы на них дают более полное представление о том, чего от нас хотят, устраняют недопонимания в нашей команде и выявляют потенциальные риски, которые могут повлиять на результаты. Зная больше о том, что нас просят создать, мы с большей вероятностью построим то, что нужно, с первого раза.

1.2.2. Реализация

Проверяя представления, мы получаем более четкое понимание того, что нас просят построить. Но это не означает, что в итоге мы получим продукт, который соответствует ожиданиям. Именно поэтому мы также тестируем реализацию, чтобы узнать следующее:

- Соответствует ли продукт ожиданиям?
- В чем продукт может не оправдать ожиданий?

Обе цели одинаково важны. Мы хотим быть уверены, что создали правильный продукт, но побочные эффекты, такие как странности поведения и уязвимости, будут существовать всегда. На примере задачи о представлении результатов поиска мы могли бы не только проверить, что функция выдает их в требуемом порядке, но и спросить:

- Что, если я введу разные условия поиска?
- Что, если релевантные результаты различны для разных поисковых инструментов?

- Что делать, если во время поиска часть сервиса не работает?
- Что, если я запрошу результаты 1000 раз менее чем за 5 секунд?
- Что произойдет, если результатов не будет?

Выходя за рамки ожиданий, мы лучше понимаем происходящее в нашем продукте и его недостатки. Это гарантирует, что в итоге мы будем верно оценивать поведение нашего приложения и не выпустим некачественный продукт. Кроме того, если мы обнаружим неожиданное поведение, то сможем попытаться устранить ошибки или скорректируем наши ожидания.

1.2.3. Значение тестирования

Модель тестирования представлений и реализации демонстрирует, что тестирование выходит за рамки простого подтверждения ожиданий и бросает вызов нашим предположениям. Чем больше мы узнаем в процессе тестирования о том, что мы хотим построить и что мы создали, тем больше эти два круга совпадают друг с другом. А чем больше они совпадают, тем точнее становится наша оценка качества.

Удивительно, но вы уже тестируете!

Поскольку цель тестирования — понять функции продуктов и разобраться, как они должны работать, стоит отметить, что вы, вероятно, уже проводите тестирование в той или иной форме. Можно утверждать, что в любой деятельности, будь то отладка кода, загрузка API или отправка клиенту вопросов о том, как должен работать ваш API, вы чему-то учитесь. Следовательно, вы тестируете.

Именно поэтому иногда считается, что тестирование — это легкая задача. Но существует различие между ситуативным, неформальным и целенаправленным тестированиями. Мы знаем, насколько сложными могут быть продукты, и только при стратегическом подходе к тестированию это различие становится очевидным.

Команды, которые хорошо информированы о собственной работе, имеют лучшее представление о качестве своего продукта. Они также лучше осведомлены о том, какие шаги предпринять для улучшения качества (что позволяет сосредоточить внимание на конкретных рисках), как внести изменения в продукт, чтобы он лучше соответствовал ожиданиям пользователей, и какие проблемы

исправить, а какие оставить без внимания. В этом и заключается цель хорошего тестирования: оно помогает командам занять позицию, в которой они могут принимать обоснованные решения и уверенно идти по пути разработки высококачественного продукта.

1.2.4. Стратегический подход к тестированию API

Я считаю, что представленная выше модель — отличный способ описать цель и значение тестирования. Однако она может показаться несколько абстрактной. Как применить ее к тестированию API? Как будет выглядеть стратегия тестирования API при таком подходе? Одна из целей этой книги — помочь вам лучше понять эту модель. Давайте рассмотрим пример стратегии тестирования API, разработанной для проекта HMRC, участником которого я был.

Проект представлял собой сервис, позволявший пользователям искать и читать нормативные документы, а также создавать отчеты. Архитектура системы упрощенно представлена на рис. 1.3.

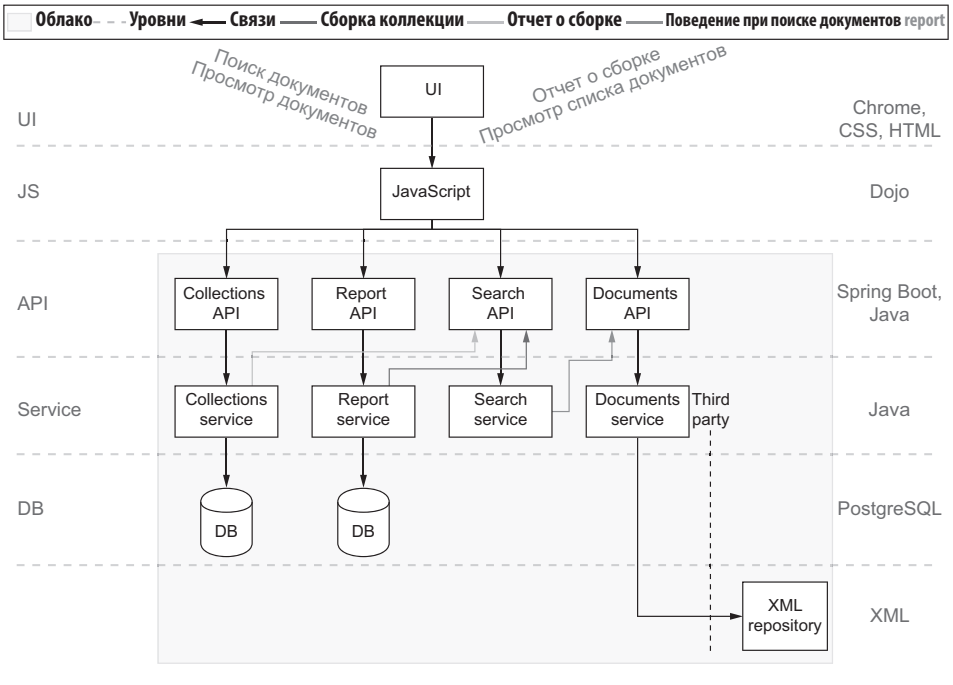

Рис. 1.3. Неформальная визуализация системной архитектуры платформы веб-API

Эта упрощенная модель дает представление о приложениях, для которых нам приходится разрабатывать стратегии тестирования API. Мы обсудим эту модель в главе 2, а здесь отметим, что приложение состоит из серии веб-API, которые предоставляют услуги пользовательскому интерфейсу и друг другу. Например, `Search` API может быть запрошен пользовательским интерфейсом и другим API, таким как `Report` API. Итак, у нас есть пример приложения, но как применить к этому контексту модель тестирования, о которой мы узнали? Опять же, лучше всего это объяснить на визуальной модели, показанной на рис. 1.4.

Рис. 1.4. Экземпляр модели описывает определенные действия по тестированию в рамках общей стратегии тестирования API

Как видим, части кругов представления и реализации заполнены операциями тестирования, которые помогут узнать, как работают наши веб-API. Для представления применимы следующие виды тестирования:

- *Тестирование дизайна API* позволяет нам подвергать сомнению идеи и улучшать общее понимание проблем, которые мы пытаемся решить.
- *Тестирование контрактов* помогает убедиться, что веб-API взаимодействуют друг с другом и корректно обновляются при внесении изменений.

Для реализации применимы такие виды тестирования:

- *Исследовательское тестирование* позволяет понять, как ведут себя веб-API, и обнаружить потенциальные проблемы.

- *Тестирование производительности* помогает лучше понять поведение веб-API под нагрузкой.

И наконец, у нас есть возможность *автоматизированных проверок* API, которые охватывают области пересечения наших представлений и реализации. Эти проверки могут подтвердить, что наши знания о работе API по-прежнему верны, и привлечь внимание к любому потенциальному ухудшению качества.

В этой книге мы узнаем больше об этих и других видах тестирования. Приведенная модель демонстрирует, что различные техники тестирования сосредоточены на разных областях нашей работы и предоставляют разную информацию. Успешная стратегия тестирования API должна быть целостной по своему подходу, сочетая множество различных техник, которые работают вместе, дополняя друг друга. Чтобы создать такую стратегию, требуется следующее:

1. *Проанализируйте контекст и его риски*: кто наши пользователи? Чего они хотят? Как работает наш продукт? Как мы работаем? Как они понимают качество?
2. *Оцените доступные типы тестирования*: знаем ли мы, как эффективно использовать автоматизацию? Знаем ли мы, что можем тестировать идеи и дизайны API до начала написания кода? Как мы можем получить пользу от тестирования в продакшене?
3. *Используйте знание контекста, чтобы выбрать правильные действия по тестированию*: какие риски наиболее важны для нас и как их снизить с помощью тестирования?

В этой книге мы рассмотрим все три пункта, чтобы дать необходимые навыки и знания для определения и реализации стратегии тестирования, которая будет помогать вам, вашей команде и вашей организации. По мере чтения книги мы будем возвращаться к общей модели тестирования, чтобы понять, какие техники тестирования работают лучше всего и где, а затем интегрировать их в эффективную стратегию тестирования. Прежде чем углубиться в многочисленные аспекты тестирования API, давайте познакомимся с несколькими подходами, которые помогут быстро изучить платформы веб-API.

ИТОГИ

- Веб-API содержат несколько уровней. Каждый из них выполняет собственные сложные задачи, которые становятся еще более сложными при их комбинировании.

- Сложность возрастает еще больше, когда несколько веб-API работают вместе в рамках сервисов для конечного пользователя на платформе.
- Понимание этой сложности является ключом к созданию высококачественного продукта.
- Чтобы достичь такого понимания, требуется целенаправленная стратегия тестирования.
- Тестирование можно рассматривать как сосредоточение внимания на двух областях: представлений и реализации.
- Мы проверяем представления, чтобы узнать больше о том, что хотим построить. Мы проверяем реализацию, чтобы узнать больше о том, что уже создали.
- Чем больше мы знаем о представлении и реализации, тем больше они пересекаются и тем лучше мы информированы о качестве нашего продукта.
- Модель тестирования показывает, как тестирование работает в областях представлений и реализации.
- Успешная стратегия тестирования состоит из множества действий, которые работают вместе для поддержки команды.

2 Начинаем наш путь тестирования

В этой главе

- ✓ Знакомство с продуктом
- ✓ Настройка продукта
- ✓ Разбираемся, что мы тестируем, собираем сведения о продукте
- ✓ Создаем визуальную модель, чтобы делиться информацией

Представим, что это наш первый рабочий день в проекте. Мы присоединились к команде и должны внедрить стратегию тестирования, чтобы помочь коллегам и улучшить качество продукта. С чего мы начнем или как будем продвигать уже имеющуюся стратегию тестирования? Какие инструменты и методы мы должны использовать?

В следующих главах мы узнаем о различных способах тестирования веб-API и о том, как разработать стратегию тестирования API, а для облегчения обучения попрактикуемся на примере продукта. Как и в тот воображаемый первый день, будем решать задачу разработки стратегии тестирования API для приложения и контекста, с которым мы не знакомы. В этой главе мы узнаем о продукте, который будем тестировать на протяжении всей книги, а также о том, как начать путь к созданию успешной стратегии тестирования API.

Подготовка

Прежде чем приступить к этой главе, я настоятельно рекомендую загрузить и установить платформу API песочницы — restful-booker, которую мы будем использовать в этой главе и далее. Подробности о том, как установить приложение, вы найдете в приложении в конце книги.

2.1. ЗНАКОМСТВО С ПРОДУКТОМ

Приступая к тестированию, целесообразно сперва изучить историю продукта. Не пожалейте времени на то, чтобы узнать о пути, который прошли команда и продукт, понять, как работает продукт, чего команда надеется достичь и какие проблемы пользователей пытается решить. Все это поможет сформировать видение стратегии тестирования.

Уже работаете над продуктом?

Хотя в этой главе описана ситуация, в которой мы начинаем новый проект без стратегии тестирования, многие из нас, скорее всего, будут работать над существующими проектами. Однако инструменты и приемы, которые мы изучим в этой главе, пригодятся независимо от контекста. Эта глава поможет научиться разбираться, как работает платформа API, и делиться этим пониманием с другими.

Для начала давайте познакомимся с платформой restful-booker и узнаем, что она может, почему была создана и что мы собираемся сделать, чтобы помочь улучшить ее качество.

2.1.1. Знакомство с нашей API-песочницей

Давайте представим, что платформа restful-booker — это реальный продукт, за который мы несем ответственность. В нашей ролевой игре платформа restful-booker была создана для владельцев отелей типа bed-and-breakfast (B&B) для управления своими веб-сайтами и бронированием. Ее функции:

- создание бренда для продвижения B&B;
- добавление номеров для бронирования с подробной информацией для гостей;
- создание бронирований гостями;

- просмотр отчетов по бронированиям для оценки доступности;
- отправка гостями сообщений хозяевам B&B.

Платформа изначально создавалась как хобби-проект для одного владельца B&B, но с тех пор выросла и теперь используется несколькими владельцами B&B для бронирования. Проект постепенно растет как по числу отелей, так и по клиентской базе, но в последнее время стали возникать проблемы. Некоторые владельцы B&B недовольны ошибками, простоями и неправильно реализованными функциями. Наша цель — предоставить стратегию тестирования, которая поможет команде улучшить качество платформы restful-booker и гарантирует, что как владельцы B&B, так и гости будут довольны продуктом.

2.2. ЗНАКОМСТВО С ПЛАТФОРМОЙ RESTFUL-BOOKER

Прочитав короткую историю платформы restful-booker, мы узнали следующее:

- У нас есть два разных типа пользователей: гости и менеджеры B&B, что следует учитывать при разработке наших API.
- Приложение имеет несколько различных функций, и это подразумевает, что несколько сервисов, вероятно, обрабатываются несколькими API.
- Ядро продукта написано на Java, что определяет, какие языки и инструменты мы могли бы использовать для автоматизации некоторых наших тестов.

Но самое главное, мы узнали, что от нас требуется разработать и внедрить стратегию тестирования API, которая поможет нашей команде улучшить качество продукта.

Мы могли бы сразу попробовать использовать знакомые методы и инструменты или просто начать отправлять запросы в различные веб-API на нашей платформе, чтобы посмотреть, что произойдет. Это может принести некоторую пользу, но не приблизит создание эффективной стратегии тестирования API. Хорошая стратегия требует понимания целей. Без знания того, как работает наша система, как она реализована, кем и для кого она создается, мы не определим стратегию.

Прежде чем мы начнем принимать поспешные решения, нужно понять продукт и проект. Это подразумевает изучение различных источников информации и использование инструментов, помогающих узнать больше о продукте, для которого мы собираемся построить стратегию. Важно отметить, что нет различия в том, что исследовать в первую очередь. В зависимости от наших собственных предпочтений или стиля обучения мы можем начать с чтения документации

или исходного кода, попросить кого-либо из команды провести демонстрацию или поиграть с продуктом. Что бы мы ни выбрали, важно помнить две вещи. Во-первых, цель состоит в том, чтобы расширить наше понимание контекста, для которого необходимо разработать стратегию. То есть внимание нужно сосредоточить на обучении, а не на доведении систем до предела для поиска проблем (хотя иногда обнаружение проблем происходит естественным образом). Во-вторых, мы должны изучить все аспекты нашего продукта и проекта. Чем больше мы узнаем, тем более ясным станет выбор стратегии, когда дело дойдет до ее реализации. Однако нам все-таки нужно выбрать, с чего начать. Поэтому давайте начнем наше исследование с рассмотрения самого продукта.

2.2.1. Изучение продукта

Поскольку в этой книге основное внимание уделяется тестированию API, мы не будем уделять слишком много внимания тестированию пользовательского интерфейса. Однако это не означает, что мы не можем использовать его в своих исследованиях. Используя наш продукт так, как это сделал бы пользователь, мы можем больше узнать как о потребностях пользователей, так и о том, как продукт их удовлетворяет.

Упражнение

Найдите время, чтобы забронировать номер и связаться с отелем B&B в качестве гостя. Кроме того, попробуйте войти в систему как менеджер B&B. Создайте комнаты для гостей, проведите ребрендинг, прочитайте отчеты и получите доступ к сообщениям. Учетные данные для входа в систему для администратора по умолчанию можно найти в файле README. Доступ к платформе restful-booker вы можете получить через http://localhost:8080 или https://automationintesting.online, в зависимости от того, запускаете вы приложение локально или нет. Делайте заметки о том, что вы узнали.

В книге по тестированию API есть также тестирование UI!

Вы уже заметили, что наша песочница поставляется с пользовательским интерфейсом, потому что платформа restful-booker используется не только для обучения тестированию API. В этой главе вы убедитесь, что если у платформы API есть пользовательский интерфейс (UI), его можно использовать для изучения работы приложения. Но в целом тема пользовательского интерфейса выходит за рамки этой книги. Если вам требуется стратегия тестирования как API, так и UI, потратьте время на изучение тестирования UI.

Инструменты разработчика

Навигация по платформе restful-booker помогает получить некоторые первоначальные знания о продукте, но наша цель — веб-API, которые находятся за пользовательским интерфейсом. Мы можем глубже изучить продукт, используя встроенные инструменты для браузеров, таких как Google Chrome и Firefox. Эти инструменты разработчика поставляются с функциями мониторинга HTTP-трафика, что позволяет нам захватывать запросы из браузера и ответы на них. Трафик доступен для анализа, чтобы понять, какие веб-API вызываются и какие данные передаются.

Давайте посмотрим на целевую страницу платформы restful-booker. Сначала откройте инструменты разработчика (самый быстрый способ — щелкнуть правой кнопкой мыши на странице и выбрать Inspect Element), а затем вкладку Network. Щелкните на фильтре XHR, после чего посетите либо https://automationintesting.online, либо http://localhost:8080.

Сокращение XHR означает XMLHttpRequests — HTTP-запросы, отправляемые из браузера в API в фоновом режиме. Их можно использовать в ситуациях, когда мы хотим асинхронно изменять данные на стороне пользователя или на стороне сервера без обновления всей страницы. Например, запрос XHR к `/branding/` можно использовать для обновления изображений и деталей домашней страницы без необходимости обновления всей страницы.

Сетевая панель покажет результаты, подобные показанным на рис. 2.1.

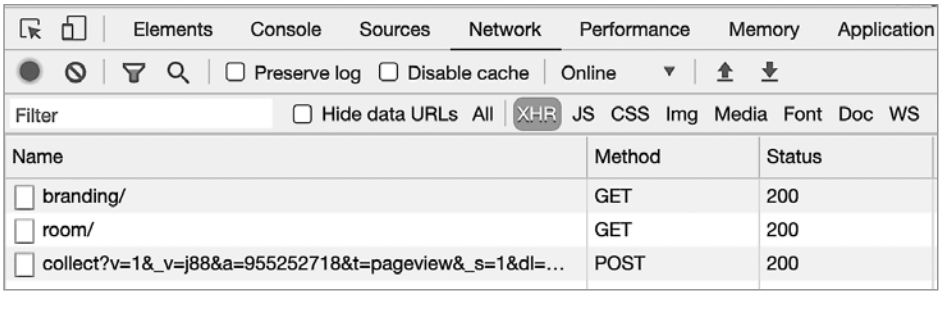

Рис. 2.1. Список HTTP-запросов во вкладке Network инструментов разработчика после вызова домашней страницы

Из результатов видно, что было сделано как минимум два разных вызова: один в `/branding/`, а другой в `/room/`. Мы можем открыть каждый вызов и посмотреть, какая информация отправляется из этих двух веб-API в пользовательский

интерфейс (UI), то есть какие изображения и текст выводятся на целевой странице. Таким образом, начальное исследование с помощью открытых инструментов разработчика может предоставить много полезной информации, которую мы можем использовать в дальнейшем.

Упражнение

Перейдите на каждую из страниц, которые мы определили в предыдущем упражнении, отслеживая трафик между браузером и веб-API платформы restful-booker. Запишите информацию о трафике, URI в HTTP-запросах, которые сообщают, какие веб-API может использовать платформа restful-booker. Посмотрите, вызываются ли другие API и каковы их URI. Также следите за тем, какие типы методов HTTP используются.

СОВЕТ Очистка истории запросов после каждого вызова упрощает просмотр новых сетевых вызовов. Вы можете сделать это, нажав кнопку Clear в верхнем левом углу рядом со значком записи.

HTTP-клиенты

Определив HTTP-трафик между пользовательским интерфейсом и веб-API, мы можем продолжить разбираться в поведении каждого веб-API. Для этого мы будем использовать Postman — клиент тестирования HTTP.

Бесплатную версию инструмента можно загрузить по адресу https://www.postman.com/downloads/. Она содержит все функции, необходимые для нашего исследования. Если вы впервые работаете с Postman, убедитесь, что вы установили инструментарий, создали учетную запись и новое рабочее пространство. После установки и настройки мы можем начать копировать HTTP-запросы, которые обнаружили в инструментах разработчика, в Postman. Рассмотрим пример такой операции для одной из конечных точек `room`:

1. Откройте инструменты разработчика и заполните вкладку Network HTTP-запросами, обновив или загрузив панель администратора.
2. Щелкните правой кнопкой мыши HTTP-запрос `/room/` и выберите Copy > Copy as Curl (выберите вариант Bash, если вам предлагается выбрать между Cmd и Bash).
3. Теперь, когда ваш HTTP-запрос скопирован как запрос curl, перейдите в Postman и нажмите Import в верхнем левом углу.

4. Когда появится всплывающее окно импорта, выберите Raw text, скопируйте curl-запрос и нажмите Continue > Import. Теперь HTTP-запрос появится в Postman.

В Postman мы можем изменить и выполнить запрос, чтобы узнать больше о том, как ведет себя веб-API. Например, с конечной точкой `GET /room/` мы могли бы сделать следующее:

- Измените URI на `/room/1`, чтобы обнаружить дополнительную конечную точку, которая показывает конкретные сведения о комнате (`room`).
- Измените метод HTTP на `OPTIONS`, чтобы в заголовках ответов в разделе Allow (заголовки ответов можно посмотреть, выбрав вкладку Headers в нижней половине окна) появились другие методы HTTP, которые можно вызывать в `/room/`.
- Найдите в заголовках запроса пользовательские файлы cookie (в частности, упоминание `token` в содержимом поля cookie).

Изменив HTTP-запрос, мы можем получить ценную новую информацию. Пока наша задача — не провести комплексное тестирование, а научиться чему-то новому.

Упражнение

Скопируйте HTTP-запросы, которые вы определили с помощью инструментов разработчика, в Postman. Создав запросы в Postman, попробуйте изменить URI, тело запроса и заголовки, чтобы узнать больше о конечных точках в каждом API и их поведении. Помните, что цель на данный момент — учиться, а не находить ошибки.

В качестве бонуса узнайте, как создавать коллекции в Postman. Добавляйте в них свои запросы для использования в будущем. На веб-сайте Postman есть подробная документация о том, как работают коллекции.

Proxy-инструменты

До сих пор наше исследование было сосредоточено на трафике между браузером и серверными веб-API. Однако не весь трафик платформы API проходит между браузером и бэкендом. Нам нужно расширить исследование, чтобы разобраться, сколько веб-API существует на платформе restful-booker и как они взаимодействуют друг с другом. Это можно сделать с помощью прокси-инструмента Wireshark.

Wireshark — это продвинутый инструмент сетевого анализа, который может перехватывать трафик на основе широкого спектра сетевых протоколов. Мы будем использовать Wireshark для мониторинга локального HTTP-трафика между API на платформе restful-booker. Поскольку при этом требуется доступ к локальному трафику, нам понадобится платформа restful-booker, работающая локально. Нам также необходимо загрузить и установить Wireshark (https://www.wireshark.org/).

Запустим Wireshark и в списке Capture выберем элемент с *Loopback* в заголовке, подобный показанному на рис. 2.2.

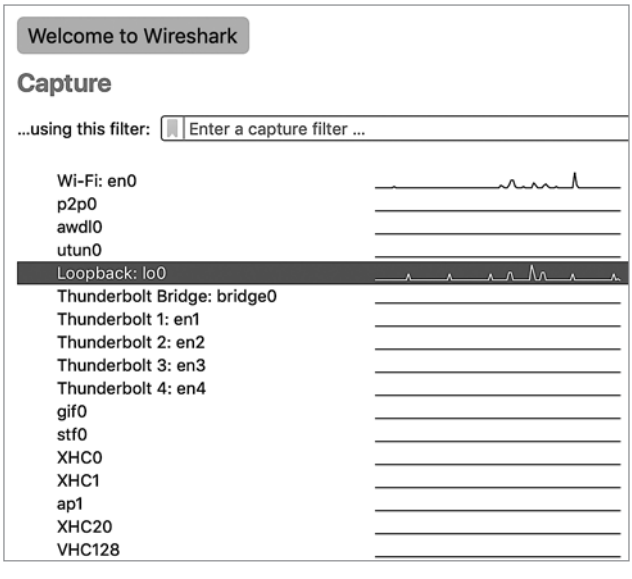

Рис. 2.2. Выбор сети Loopback из списка в Wireshark

Если вы не видите Loopback в списке, вероятно, вам требуются дополнительные разрешения или плагины для ОС.

Wireshark может отслеживать любую сетевую активность внутри нашей машины. После выбора интерфейса Loopback мы увидим список всех сетевых запросов и ответов, которые были запущены с момента начала мониторинга Wireshark. Список будет быстро заполняться сетевыми действиями из разных протоколов (HTTP, TCP, UDP и т. д.). Чтобы немного облегчить наше исследование, введите `http` в параметрах фильтра и нажмите Enter. Список обновится и покажет только HTTP-запросы и ответы.

Устранение неполадок Wireshark

Если у вас возникли проблемы с Wireshark, то выполните несколько проверок.

- Если вы используете Wireshark на Mac, убедитесь, что вы также установили ChmodBPF, что необходимо для анализа трафика на локальном узле. Подробности установки можно найти по адресу https://formulae.brew.sh/cask/wireshark-chmodbpf.
- Не все сетевые карты поддерживают возможность мониторинга внутреннего трафика. Если вам не удается настроить Loopback, попробуйте выяснить, работает ли Wireshark с вашей сетевой картой.

Теперь, когда Wireshark настроен, мы можем отслеживать сетевую активность между различными веб-API. Например, из предыдущих операций мы узнали, что существует конечная точка `POST /room/`, которую мы можем использовать для создания комнаты. Если мы используем Postman для создания комнаты, отправив POST-запрос `/room/` к API `room` (комнаты), как показано на рис. 2.3, то в Wireshark увидим запрос, сделанный не только к `/room/`, но и к другому локальному API `localhost:3004/auth/validate`, как показано на рис. 2.4.

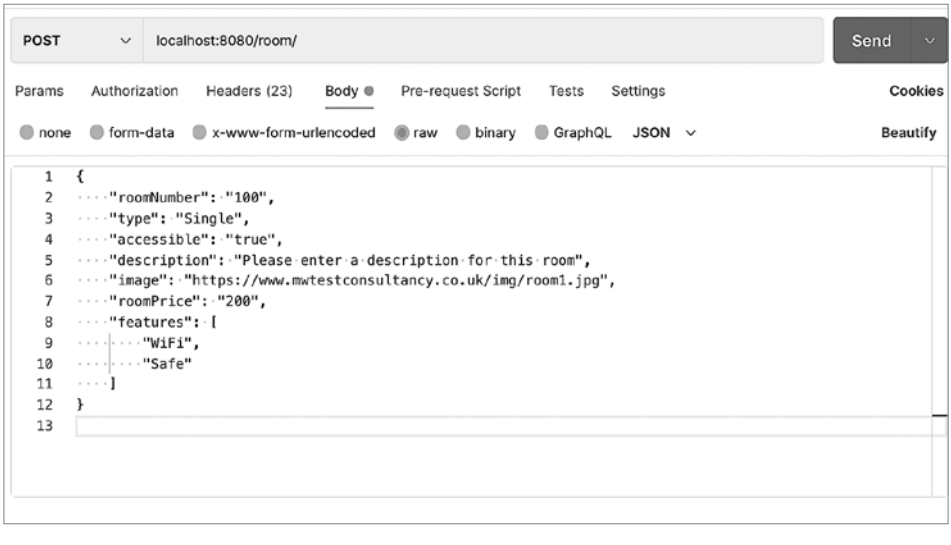

Рис. 2.3. HTTP-запрос POST для API с использованием Postman, который позволяет создать комнату

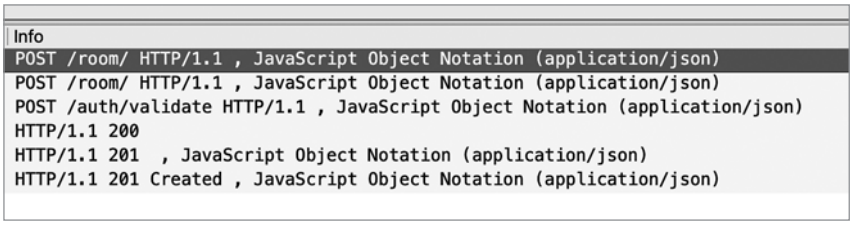

Рис. 2.4. Пример HTTP-запроса, перехваченного Wireshark, с выделенным POST-запросом к /room/

На основании этой информации мы можем сделать два вывода:

- Существует веб-API `auth`, который прослушивает порт 3004.
- `room` API отправляет запросы на авторизацию.

Это демонстрирует, как Wireshark помогает узнать больше о платформах API и о связях веб-API друг с другом. Однако это работает, только если есть возможность прослушивать сетевые устройства на компьютере. Если платформа API развернута в другом месте и у нас нет локального доступа к ней, такая возможность недоступна.

Упражнение

Настройте Wireshark для отслеживания трафика локального узла и отображения запросов, которые отправляются между веб-API. После настройки запускайте запросы, перехваченные в Postman, и наблюдайте, какой трафик инициируется между веб-API. Например, попробуйте вызвать `GET /report/` и понаблюдайте за прочими запросами, которые `report` API отправляет другим API. Запишите результаты и при желании обновите свою коллекцию Postman.

С помощью нескольких простых действий и инструментов мы обнаружили, что платформа restful-booker состоит из набора веб-API, которые выполняют различные функции, такие как создание комнат, составление отчетов и обеспечение безопасности продукта. Исследуя продукт, мы получили начальное понимание того, как он работает, и, возможно, даже начали обдумывать определенные варианты стратегии. Однако давайте рассмотрим некоторые дополнительные источники информации, которые позволят еще больше расширить наше представление.

2.2.2. Исследования за пределами продукта

Платформа restful-booker имеет пользовательский интерфейс с набором функций для взаимодействия. Однако не все проекты будут иметь пользовательские интерфейсы или находиться в завершенном состоянии для анализа. Тогда потребуются дополнительные источники информации. К счастью, при создании программного обеспечения появляются дополнительные материалы, такие как документация, пользовательские истории и исходный код, которые могут служить нам отличным руководством. Давайте рассмотрим несколько примеров для платформы restful-booker, чтобы продемонстрировать разные типы дополнительной информации, которые мы можем использовать.

Документация

Хотя отношение к документации менялось по мере того, как agile стала доминирующей методологией разработки ПО, маловероятно, что у проекта вообще не будет документации. Для многих проектов времена создания объемных подробных документов с описанием всех требований, вероятно, прошли. Однако неформализованная документация, такая как wiki, документация к API или пользовательские истории, поможет лучше понять продукт.

Например, с платформой restful-booker мы можем многое узнать из файлов README проекта, которые можно найти по адресу http://mng.bz/nNna. Там приводятся уже известные нам сведения о том, на каких версиях Java и NodeJS работает продукт и как его запустить локально. Но файл README также сообщает нам, что каждый API в рамках проекта имеет свою собственную документацию. При открытии исходного кода одного из модулей API (подробнее об этом чуть ниже) открывается дополнительная документация, включая README, которую можно найти по адресу http://mng.bz/v6g7.

Эта документация содержит более подробные сведения о том, как создавался API, как его можно настроить и запустить, а также информацию о дополнительных документах.

Как мы узнаем из следующей главы, современные средства разработки API предлагают возможности для создания богатой интерактивной документации, например такой: http://mng.bz/44dw. Откройте ее, и вы не только обнаружите аккуратно составленный список всех конечных точек веб-API `room`, но и получите возможность взаимодействовать с ними.

Например, открытие `room-controller` показывает список запросов, которые мы можем использовать. Выбор `GET /room/` дает подробное представление о том, как должен быть построен HTTP-запрос для конечной точки и что возвращается

в ответе. Нажав кнопку Try it, мы получаем кнопку Execute, которая отправит пример запроса к веб-API и покажет ответ. Это сочетание подробных инструкций о работе каждой конечной точки и возможность создавать и отправлять запросы бесценны для глубокого изучения наших веб-API. Предоставляется возможность исследовать продукт с дополнительным контекстом и полезными рекомендациями.

Мы можем больше узнать об истории наших продуктов, просмотрев также пользовательские истории, файлы функций и документы с требованиями в инструментах управления проектами. Качество материалов будет зависеть от того, как построена работа в команде и как сохраняется информация. В некоторых случаях это список выполненных работ в Jira, в других может потребоваться просмотр переписки. Но изучение этих материалов позволит узнать дополнительный контекст:

- Потенциальные возможности, которые мы упустили при анализе продукта.
- Сотрудники, ответственные за разработку функций.
- Пользователи и проблемы, которые мы надеемся решить.
- Путь, который прошел продукт по мере развития и изменения.

Например, просмотр доски проектов платформы restful-booker на GitHub (http://mng.bz/QvGG) позволит узнать следующее:

- Ошибки, с которыми сталкивалась платформа в прошлом.
- Технический долг, раскрывающий детали реализации.
- Пользовательские истории о том, как должна работать каждая функция.

Примечательно разнообразие сведений. Например, из пользовательских историй мы можем получить больше информации о функциях платформы restful-booker, а из карт технического долга можем узнать технические подробности, например, какие используются библиотеки, инструменты и база данных.

Исходный код

Кому-то может показаться, что просмотр исходного кода является наиболее очевидным и простым способом изучения работы продукта или платформы. Кого-то эта идея обеспокоит. Реакция на нее зависит от опыта работы с кодом. Если вы сами привыкли писать код, то просмотр чужого исходного кода будет естественным шагом. Если такой привычки нет, то не будет и желания знакомиться с исходным кодом. Однако даже если вы относитесь ко второй группе, есть несколько подходов, чтобы использовать эту возможность.

Прежде всего, важно помнить, что наша текущая цель — познакомиться с платформой restful-booker. Поэтому нужно понять существующий код, а не писать новый и придумывать свежие идеи. Чтение и написание кода — два совершенно разных действия. Чтобы писать код, нужно понимать язык и то, как его использовать для решения проблем. Когда мы читаем код, мы придаем смысл принятому решению. Вместо того чтобы решать проблемы, мы ищем следующую информацию:

- Названия модулей, например имена API в корне кодовой базы, которые регулярно появляются в других частях кодовой базы (например, `room`, `booking`, `branding` и т. д.).
- Пакеты и имена классов, которые указывают на поведение каждого API. Например, в `room` веб-API есть класс `AuthRequests`. Такое имя убедительно свидетельствует о том, что оно обменивается данными с другим веб-API, `auth`.
- Файлы зависимостей, такие как pom.xml и package.json (имя определяется языком), позволяют узнать, какие типы библиотек и технологий используются.
- Комментарии к коду, которые описывают работу методов или функций.

Это всего лишь несколько примеров деталей, которые вы можете почерпнуть из обзора кодовой базы, и это не требует глубоких знаний языка, на котором построены веб-API. Помогает ли такое глубокое знание? Да, помогает. Но всем нам нужно с чего-то начинать, и мы можем развить свои навыки, знания и уверенность в себе, потратив время на изучение кодовых баз.

Беседа с членами команды

Если команда проекта не была полностью заменена, а предыдущие ее участники не были унесены призраками, то непременно найдутся люди с ценными знаниями, и с этими людьми следует поговорить. Вспомните нашу модель тестирования из главы 1. Необходимо узнать о следующем:

- *концепция* — что мы хотим создать;
- *реализация* — что уже создано.

Мы можем получить сведения в этих двух областях, поговорив с командой.

Концепция

Большая часть того, что мы обсуждали до сих пор, относится к пониманию принципов работы продукта и его поведения, но также важно понимать, для кого мы создаем наш продукт и почему. Поэтому целесообразно поговорить с людьми,

чьи роли больше связаны с концептуальной частью, такими как владельцы продукта, дизайнеры и бизнес-аналитики. Информация, которой они поделятся, не только расширит наши знания о продуктах, но и поможет сформировать стратегию тестирования.

Например, о платформе restful-booker мы узнаем, что надежность и время безотказной работы сайта имеют большое значение для владельцев наших B&B. Соответственно, в стратегии тестирования мы можем сделать приоритетным снижение рисков, связанных с надежностью. В следующей главе мы рассмотрим этот подход более подробно на основе данных о пользователях и их проблемах.

Реализация

Ранее мы обсуждали, что кодовая база может быть ценным источником информации. Но важно помнить, что она — результат работы членов команды, которые имеют богатый опыт и знания в разработке. Вы можете получить эту ценную информацию, обсудив с конкретными людьми их работу. Можно использовать форматы неформальной беседы, демонстрации продукта или совместного просмотра кода.

2.3. ФИКСАЦИЯ НАШЕГО ПОНИМАНИЯ

По мере того как растет понимание платформы веб-API, становится ясна сложность продукта. Мы можем делать заметки и пытаться запомнить различные аспекты наших API-платформ, но в какой-то момент нужно упорядочить информацию в форме, которая поможет осмыслить то, что мы узнали, а также донести это до других. Можно ли сделать это так, чтобы не получить массу дополнительной документации и не запутаться, вернувшись к изучению после недельного отпуска? Целесообразно создать визуальную модель продукта, отражающую наши знания в краткой и понятной форме, которую можно легко обновить.

2.3.1. Сила моделей

Представьте, что вы собираетесь отправиться на автомобиле в город, в котором никогда раньше не были. Что вы будете делать? Скорее всего, вы откроете какую-либо карту, бумажную или в приложении, чтобы проложить маршрут до этого города, а также прикинуть, сколько времени займет поездка и возможные остановки. Таким образом, вы используете модель для решения проблемы навигации.

В примере выше моделью является карта, и она демонстрирует, как мы используем модели для осмысления окружающей нас информации. На карте, скорее всего, будут отмечены основные дороги, но не отображены другие детали, такие как рельеф местности. Наша карта не является моделью топографии маршрута, по которому мы хотим проехать, но она содержит важную информацию, необходимую нам для определения направления движения. Ведь именно так и задумано.

Возможно, вы сталкивались с афоризмом «Все модели ошибочны, но некоторые из них полезны». Это отличная фраза, которая отражает ценность хорошей модели. Хорошая модель предоставляет информацию, которая важна для нас, удаляя при этом другую информацию, которая нам не нужна. Если мы осознаем, что модели ошибочны по своей природе, то можем использовать это в своих интересах. Мы можем создавать модели, которые предназначены для усиления конкретной информации, полезной для нас, игнорируя при этом другие элементы. Например, мы можем построить диаграмму архитектуры системы, которая покажет конкретные детали платформы API, чтобы помочь понять стратегию тестирования, как показано на рис. 2.5.

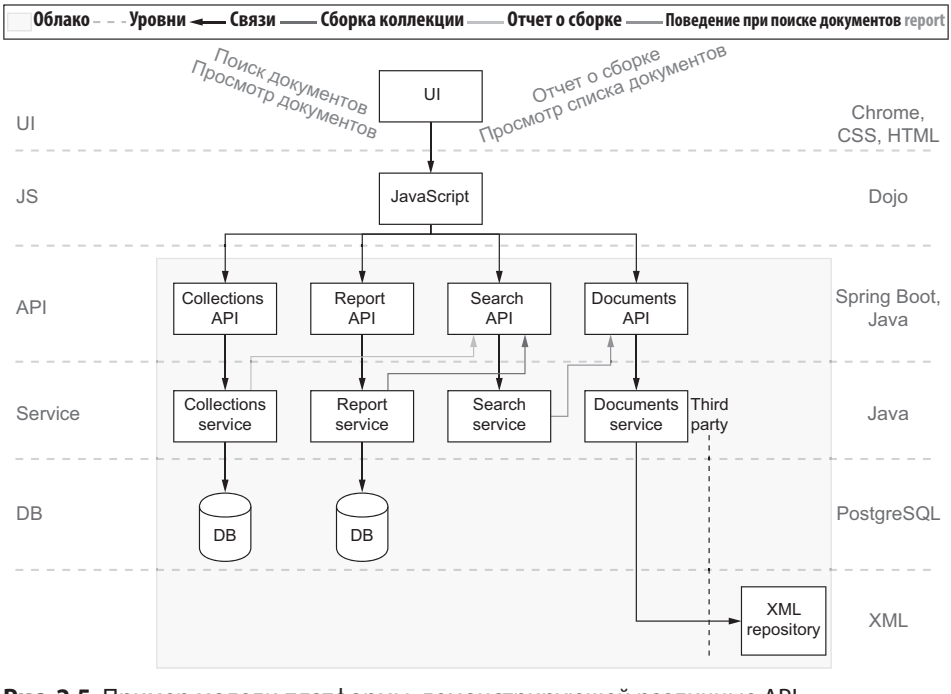

Рис. 2.5. Пример модели платформы, демонстрирующей различные API и их взаимосвязи

2.3.2. Построение собственных моделей

Построение диаграмм архитектуры системы — не новая концепция. Многие команды используют различные их типы, такие как диаграммы последовательности, состояния или системные. Но мы должны помнить, что разные подходы к построению диаграмм — это модели, которые должны передать конкретную информацию и решить конкретные проблемы. То, как мы моделируем наши системы, будет влиять на принятие решений. Цель нашего упражнения по моделированию заключается не в том, чтобы вписать полученные знания в уже существующий образ мышления. Мы должны достичь следующего:

- Осмыслить полученную информацию.
- Определить задачи и возможности тестирования.
- Получить дополнительные сведения от других людей.

Такой подход означает, что мы можем свободно излагать свои мысли в графической форме, чтобы передать важную для нас информацию.

Итак, давайте создадим новую модель платформы restful-booker, которая отражает уже полученную информацию. Мы можем воспользоваться инструментами для моделирования продуктов, такими как Visio, Miro и diagrams.net. Ниже мы будем использовать diagrams.net для создания начальной модели, но вы сами можете выбрать, какой инструмент более удобен. Если хотите, поэкспериментируйте с несколькими или воспользуйтесь старыми добрыми бумагой и ручкой.

Давайте начнем создавать нашу модель, зафиксировав информацию, которую мы уже выяснили о платформе restful-booker. На рис. 2.6 визуализированы пользовательский интерфейс и бэкенд.

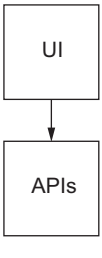

Рис. 2.6. Начальная модель, которая помогает разделить систему на две основные области: пользовательский интерфейс и бэкенд

Хотя эта модель довольно общая, она показывает два основных раздела платформы restful-booker и связь между ними. Затем мы узнаем, что наш бэкенд содержит

несколько различных веб-API, таких как `room` и `auth`. Мы можем расширить нашу модель до показанной на рис. 2.7, которая более подробно описывае бэкенд.

Теперь мы видим, что платформа restful-booker имеет несколько веб-API. Но пока отсутствует связь, в которой `room` API спрашивает `auth` API, можно ли создать комнату. Мы можем представить это, расширив предыдущую модель до показанной на рис. 2.8.

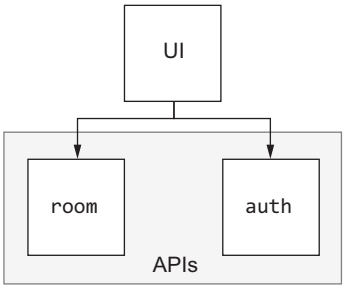

Рис. 2.7. Расширенная модель, которая показывает, что бэкэнд состоит из нескольких API

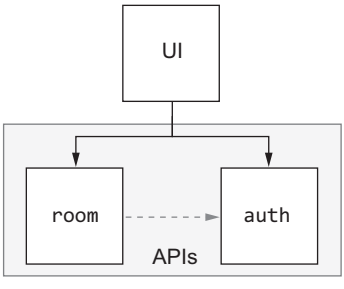

Рис. 2.8. Расширенная модель, которая показывает взаимосвязь между двумя внутренними API

В этом небольшом фрагменте модели платформы restful-booker визуально определено следующее:

- Тестирования требуют несколько веб-API.
- `room` зависит от `auth`, поэтому `auth` может иметь приоритет в очередности тестирования.
- Наши веб-API должны иметь возможность отправлять запросы и получать ответы друг от друга.
- Пользовательский интерфейс и API должны иметь возможность передавать информацию друг другу.

По мере расширения нашей модели благодаря новой информации мы начинаем естественным образом понимать риски, которые мы захотим устранить, и возможности для тестирования.

Итерация моделей и получение обратной связи

Необходимо отметить, что наша модель была построена итерационным способом. Модели — это отличный способ продемонстрировать свое представление

другим людям, чтобы получить от них обратную связь и новые знания. Интерпретируя информацию, мы строим в голове разные модели, которые может быть трудно объяснить другим. Когда вы пытаетесь объяснить информацию, организованную определенным образом в вашей голове, другой человек старается вписать эти сведения в свою собственную модель мышления, что непросто и может привести к непониманию. Представляя визуальную модель лично или в документации, вы делитесь не только своими знаниями, но и их интерпретацией. Тогда другому человеку легче понять, из чего вы исходите, и сформировать обратную связь таким образом, чтобы она имела смысл для вас. Кроме того, это упрощает взаимопонимание, поскольку ваши модели проще совместить. Вот почему итеративное создание и обмен моделями с другими людьми могут быть очень полезными. Мы учимся вместе, узнаем больше и получаем более глубокое понимание.

Упражнение

Теперь, когда мы лучше понимаем, как работают модели, и рассмотрели несколько примеров их построения, давайте вернемся к нашей первоначальной задаче: разобраться, как функционирует платформа restful-booker. На протяжении всей книги мы будем изучать различные виды тестирования, которые можно использовать для снижения рисков. Но прежде чем включить их в стратегию тестирования, нужно понять, как работает restful-booker. Как мы уже обсуждали, лучший подход — создание модели.

Итак, узнайте как можно больше о платформе restful-booker, а затем попытайтесь объединить то, что вы узнали, в визуальную модель. Как минимум создайте модель, отражающую взаимодействие различных веб-API платформы и конечные точки, которые они могут содержать. Помните, что модель поможет понять смысл того, что мы собираемся тестировать, поэтому организуйте ее так, как вам удобно.

2.4. ПОЗДРАВЛЯЮ, ВЫ УЖЕ ТЕСТИРУЕТЕ!

Проведенное исследование позволило нам продвинуться по пути разработки стратегии тестирования веб-API. Прежде чем завершить эту главу, подумайте о проделанной работе. Из главы 1 вы узнали, что цель тестирования — разобраться в том, что мы создаем сейчас и что хотим создать в будущем. Мы также обсудили, что любая деятельность, которая помогает нам узнать о том, что мы создаем, может считаться тестированием, главное, делать это целенаправленно.

Проведенные эксперименты с продуктом и построение визуальных моделей показывают, что для начала тестирования требуется не так уж много. С помощью нескольких простых инструментов и активного поиска новой информации мы можем многому научиться. Однако тестирование легко освоить, но трудно развить. Наши исследования помогли понять работу платформы restful-booker, но пока нашему подходу к тестированию не хватает продуманности и организованности.

Время на тестирование всегда будет ограниченно. Поэтому хотя изученное в этой главе служит отправной точкой, нам потребуется более продуманный и целенаправленный подход. Нам нужна четкая, эффективная стратегия тестирования. Очень заманчиво просто начать тестировать. Но только определение целей и планов по их достижению позволит провести тестирование для заданного контекста так, чтобы помочь нашим командам создать качественный продукт.

ИТОГИ

- В качестве примера для изучения тестирования мы будем использовать платформу restful-booker. Она доступна онлайн и как локальное приложение.
- Прежде чем приступить к разработке и реализации стратегии тестирования, необходимо узнать контекст, в котором мы работаем. Для этого следует изучить продукты, сопровождающие документы, а также пообщаться с командами программистов.
- Инструменты разработчика и HTTP(S)-клиенты помогают узнать больше о платформах API и о том, как они работают.
- Разобраться в контексте позволяет также документация, изучение исходного кода и общение с членами команды.
- Полученные сведения надо организовать в стройную систему путем построения модели.
- Визуальные модели помогают определить потенциальные риски и возможности тестирования.
- Исследования, которые мы проводим, являются формой тестирования, потому что помогают лучше понять продукты и идеи. Но для достижения успеха нам необходим более системный подход.

3 Качество и риски

В этой главе

- ✓ Как использовать качество для определения целей нашей стратегии
- ✓ Что такое качество и как его определить
- ✓ Что такое риски и как они влияют на качество
- ✓ Как выявить риски с помощью различных методов
- ✓ Как риски снижаются благодаря тестированию

Теперь, когда мы лучше познакомились с нашим продуктом, платформой restful-booker, пришло время приступить к разработке стратегии. В частности, нам нужно ответить на следующие два вопроса:

- Каковы цели нашей стратегии?
- Как мы собираемся достичь этих целей?

Важно прояснить эти два момента до того, как мы начнем углубляться в рассмотрение конкретных видов тестирования, чтобы убедиться, что выполняемая работа приносит пользу. Если этого не сделать, тестирование будет в лучшем

случае непоследовательным, а в худшем — пустой тратой времени. Как говорит японская пословица, «ви́дение без действия — это мечта, а действия без видения — это кошмар». Мы хотим убедиться, что у нас есть четкая цель, чтобы можно было не только приступить к формированию стратегии, но и оценить ее успешность. В предыдущих главах мы узнали, что можно без проблем подобрать несколько видов тестирования и изучить продукт. Но ключ к успешной стратегии заключается в том, чтобы тестировать целенаправленно. Итак, чтобы ответить на наши вопросы, потребуется:

- Четкая цель для нашей стратегии. Чтобы ее определить, нужно разобраться, что означает понятие «качество» для наших пользователей.
- Список потенциальных возможностей для проверки, то есть рисков.

Представим это в виде модели с пропущенными элементами, как показано на рис. 3.1.

Цель стратегии	???????
Шаги к достижению цели	???????

Рис. 3.1. Начальная модель, которая подчеркивает необходимость определения цели стратегии и шагов для ее достижения

В этой главе мы рассмотрим оба вопроса, чтобы можно было заполнить пропуски в начальной модели и начать определять приоритеты для правильного тестирования.

3.1. ЧТО ТАКОЕ КАЧЕСТВО

Давайте начнем с разговора о качестве и о его значении в контексте разработки программного обеспечения. Для этого рассмотрим небольшую аналогию. Поищите фразы типа «10 величайших альбомов», откройте несколько первых ссылок и сравните результаты. Скорее всего, обнаружатся некоторые сходства и различия (а также несколько действительно выдающихся альбомов). Но совершенно очевидно, что найденные списки не отражают единодушное мнение. У разных людей разные вкусы. Более того, иногда люди со временем меняют свое мнение. Сравните первую десятку в списке 500 величайших альбомов всех времен, составленном журналом *«Rolling Stone»* в 2003 году, с тем же списком в 2020 году, и вы увидите заметную разницу. Например, альбом группы The Beatles *«Sgt Pepper's Lonely Heart Club Band»* необъяснимым образом опустился с первого

места на двадцать четвертое. Одно и то же издание, возможно, разные авторы, но различия в оценках качества очевидны.

Качество — это весьма субъективное и изменчивое понятие, что подтверждает следующее определение:

> *Качество — это ценность для отдельного значимого человека в определенный момент времени.*

Первая часть определения принадлежит Джерри Вайнбергу (Jerry Weinberg). Она напоминает, что люди — сложные личности с уникальным опытом, и это приводит к разным представлениям о качестве. Пользователь А может считать, что ваш продукт высокого качества, в то время как пользователь Б с этим не согласится. При этом пользователь Б может быть для нас неважен и неинтересен, поэтому Джеймс Бах (James Bach) и Майкл Болтон (Michael Bolton) добавили в определение часть «для <...> значимого человека». Продукты, которые мы создаем, удовлетворяют конкретные потребности конкретных людей, поэтому наше определение качества должно зависеть от их взглядов на качество. Мы хотим отдать предпочтение людям, которые помогают нам держаться на плаву. Наконец, мы должны знать, что представление человека о качестве зависит от контекста, который меняется со временем, поэтому Дэвид Гринлис (David Greenlees) ввел в определение ссылку на время. Отличным примером значения момента времени является концепция «ситуационного нарушения», когда способности пользователя временно снижаются из-за конкретной ситуации. Например, представьте, как меняется восприятие пользователем качества работы приложения Siri, когда он использует его в тишине своего дома и на оживленной улице.

Понимание того, что качество является изменчивым/субъективным понятием, можно использовать при определении цели стратегии тестирования. Разобравшись в понимании качества нашими пользователями, мы можем определить, что надо тестировать, а что нет.

3.1.1. Характеристики качества

У каждого человека свое представление о качестве, на которое влияют прошлый опыт, предубеждения и повседневная жизнь. Как собрать и проанализировать информацию, чтобы понять это представление и преобразовать его в набор четких рабочих целей? Мы должны найти баланс: собрать достаточно подробную информацию о качестве, чтобы идти по правильному пути, но при этом не увязнуть в деталях, чтобы не перестать видеть лес за деревьями. Этого можно достичь путем систематизации характеристик качества.

Характеристика качества — это способ высокоуровневого описания какого-либо одного аспекта качества. Например, у нас может получиться следующее:

- *Внешний вид и ощущения*. Продукт считается качественным, если он хорошо выглядит или приятен в использовании. Он может иметь хорошо продуманную компоновку или изящный дизайн, облегчающий работу с ним. Например, для продукции Apple «внешний вид и ощущения» могут быть приоритетной характеристикой качества.
- *Безопасность*. Продукт считается качественным, если он обеспечивает защиту и безопасность информации. Это может быть надежное хранение данных, определенный уровень конфиденциальности или защита от нежелательного внимания. Например, менеджер паролей должен давать пользователям уверенность в безопасности.
- *Точность*. Продукт считается качественным, если он точно обрабатывает информацию. Он должен уметь работать со сложными процессами с множеством деталей и возвращать достоверные данные. Например, врач-терапевт рассматривает точность диагностики медицинским прибором как важную характеристику качества.

Характеристики качества многочисленны, и мы должны объединять их, чтобы получать надежную картину представления наших пользователей о качестве. Например, список характеристик, составленный Рикардом Эдгреном (Rikard Edgren), Хенриком Эмильссоном (Henrik Emilsson) и Мартином Янссоном (Martin Jansson) в работе *«The Test Eye Software Quality Characteristics list»*, содержит более ста различных характеристик (http://mng.bz/XZ8v).

Анализируя отзывы пользователей, мы можем определить, какие характеристики качества для них важнее, а затем использовать эти характеристики, чтобы определить цели нашей стратегии тестирования. Это поможет увязать нашу работу с задачей создания продукта, который считается качественным. Например, пользователи услуги по заполнению личных налоговых деклараций на платформе API могут отдавать предпочтение следующим характеристикам качества:

- *Интуитивность* — пользователь хочет, чтобы процесс подачи документов был как можно более простым.
- *Точность* — пользователь хочет, чтобы платформа правильно рассчитывала сумму налога.
- *Доступность* — пользователь не хочет, чтобы платформа вдруг перестала работать в последний день подачи деклараций.

Результатом должна стать стратегия, в которой приоритет отдается тестированию параметров, связанных именно с этими характеристиками качества.

Упражнение

Подумайте о продукте, которым вы регулярно пользуетесь, и просмотрите список характеристик качества *«The Test Eye Software Quality Characteristics list»*. Выберите несколько характеристик и запишите их. После этого расположите их в порядке приоритета по своему усмотрению. Есть ли среди них те, на которые вы бы обратили внимание до знакомства с этим списком?

3.1.2. Знакомство с пользователями

Определение правильных характеристик качества для нашей стратегии тестирования требует лучшего знакомства с пользователями. Чем больше мы о них знаем, тем точнее будем выбирать характеристики качества, что, в свою очередь, сделает наше тестирование целенаправленным и более ценным. Поэтому прежде чем приступить к определению характеристик качества, мы должны потратить время на изучение наших пользователей.

Стоит отметить, что исследования пользовательской аудитории — сама по себе очень обширная тема. Если в вашей команде или организации есть люди, ответственные за этот вид работы, с ними надо сотрудничать для выявления качественных характеристик продукта. Однако даже если людей с такими навыками нет, существует несколько способов, с помощью которых вы сможете узнать больше о пользователях.

Беседы с владельцами и пользователями продукта

Самый быстрый и простой способ составить представление о качестве — это опросить наших пользователей. Будь то формальный процесс интервьюирования или случайная беседа за чашкой кофе, общение поможет лучше понять, что означает для них качество. Из личного опыта могу посоветовать построить беседу так, чтобы характеристики качества проявлялись естественным образом. Это означает, что нужно просто вести нормальный разговор, а не просить пользователя выбрать характеристики из списка. Если вы представите список, пользователь, скорее всего, скажет, что все они одинаково важны, что не очень-то поможет.

Если возможности общаться с пользователями нет, то обратная связь все-таки должна поступать в вашу команду по какому-то каналу (иначе откуда бы вы знали, что создавать?). Поэтому если мы не можем поговорить с пользователями, то

следует попытаться поговорить с их доверенным лицом или представителем. Это может быть владелец продукта, бизнес-аналитик или другая заинтересованная сторона в вашей организации. Разговор с этими людьми поможет не только определить приоритетные характеристики качества, но и донести до собеседников саму идею ценности таких характеристик в надежде, что они будут стараться изучить этот вопрос, общаясь с конечными пользователями.

Наблюдение за поведением людей

Инструменты для мониторинга поведения пользователей развились до такого уровня сложности, что теперь мы можем делать выводы о потребностях и отношении пользователей к нашим продуктам, не общаясь с ними напрямую. Информацию, полученную с помощью инструментов мониторинга, можно использовать для определения характеристик качества. Мы можем отслеживать объем трафика в наших продуктах, чтобы понять, является ли производительность приоритетной характеристикой. Например, если новая функция вызывает большой отток пользователей, мы можем проанализировать ее и выяснить, почему она способствовала оттоку. Возможно, она была слишком сложной в использовании, имела неудачный дизайн или плохо работала. На основе этого анализа мы можем определить такие характеристики, как удобство использования, привлекательность или завершенность.

Как уже говорилось, это лишь несколько возможных подходов к тому, чтобы больше узнать о наших пользователях и их представлениях о качестве. Какой бы подход мы ни выбрали, цель состоит в том, чтобы выработать модель поведения для выяснения того, что хотят наши пользователи. По мере работы наши продукты развиваются и изменяются, а вместе с ними меняются и пользователи. Это означает, что нам необходимо регулярно общаться с пользователями, чтобы знать их мнение о качестве. Выделите в своей стратегии время для «корректировки курса» и бесед с пользователями, чтобы обновлять приоритеты характеристик качества и цели стратегии тестирования.

Упражнение

Выберите одну из приведенных здесь методик и примените ее, чтобы узнать что-то новое о своей целевой аудитории. Попробуйте определить хотя бы одну или две характеристики качества.

3.1.3. Качество и определение целей для нашей стратегии

После получения информации от пользователей ее нужно проанализировать, чтобы выяснить, какие характеристики качества имеют значение, а затем

использовать их как цели для нашей стратегии. Например, хотя пользователи платформы restful-booker имеют виртуальный характер, мы можем определить несколько характеристик качества для разных групп пользователей. Для гостя могут быть важны следующие качества:

- *Завершенность* — пользователю требуются все возможности для бронирования и получения номера.
- *Интуитивность* — пользователь хочет, чтобы взаимодействие с нашими API было простым.
- *Стабильность* — пользователь не хочет, чтобы происходили сбои или терялись бронирования (в результате чего он может оказаться в отеле B&B без брони).
- *Конфиденциальность* — пользователь хочет, чтобы его данные о бронировании были в безопасности.

Для администратора могут быть важны следующие характеристики:

- *Завершенность* — администратор хочет иметь все функции для управления бронированиями и номерами.
- *Интуитивность* — администратор хочет, чтобы административные функции были просты в использовании.
- *Стабильность* — администратор не хочет, чтобы сайт падал, когда он вносит изменения в систему.
- *Доступность* — администратор хочет, чтобы сайт всегда был доступен.

Как мы видим, некоторые характеристики для разных типов пользователей совпадают, а некоторые — нет. Мы можем сесть всей командой и определить приоритеты для этих характеристик. Например, мы могли бы отдать предпочтение характеристикам, которые важны для обеих групп, затем — характеристикам, важным для гостей (поскольку они являются более многочисленной группой пользователей), и, наконец, характеристикам, важным только для администраторов. Этот список в конечном итоге позволяет определить, что целью стратегии является помощь нашей команде в улучшении следующих характеристик качества:

- интуитивность;
- завершенность;
- стабильность;
- конфиденциальность;
- доступность.

Мы можем отразить их в обновленной модели стратегии, как показано на рис. 3.2.

Цель стратегии	Характеристики качества Интуитивность | Завершенность | Стабильность | Конфиденциальность | Доступность
Шаги к достижению цели	???????

Рис. 3.2. Расширенная модель, которая показывает, как мы можем использовать качественные характеристики для определения целей нашей стратегии

Упражнение

Согласны ли вы с этими характеристиками? Поставьте себя на место гостя или администратора платформы restful-booker. Какие качественные характеристики вы бы добавили или убрали? Составьте свой список, он понадобится чуть позже.

3.2. ВЫЯВЛЕНИЕ РИСКОВ ДЛЯ КАЧЕСТВА

Характеристики качества устанавливают цели, которых мы стремимся достичь, и помогают определить направление стратегии. Теперь нам необходимо рассмотреть, какие шаги следует предпринять для достижения целей по улучшению качества. Для этого мы выявим ситуации, в которых наши характеристики качества могут быть подвержены рискам.

В контексте разработки ПО риск — это вероятность негативного влияния на качество продукта. Например, риском является возможность ошибки, которая приведет к принятию неверных данных, или это может быть риск повреждения данных. Когда мы имеем дело с рисками, мы, по сути, делаем ставку. В момент выявления риска мы не знаем, проявится ли он на самом деле и станет проблемой или же ничего не случится. Именно здесь нам помогает информация, которую мы получаем в ходе тестирования. Сосредоточив внимание на тестировании в той области, где риск может оказать определенное влияние, мы можем решить, действительно ли он является проблемой, требующей устранения.

Ключевым моментом в работе с рисками является то, что мы не знаем, является ли риск реальным, пока не проведем тестирование. В процессе создания продуктов мы выявим множество рисков (и пропустим еще больше), но у нас не будет времени на тестирование каждого из них. Поэтому мы должны быть избирательны в выборе рисков, которые тестируем и не тестируем. Таким образом, риск выступает

в качестве руководства для нашей стратегии тестирования и действий, которые мы выполняем в рамках этой стратегии. Мы выбираем риски, которые имеют наибольшее значение, и тестируем их, делая ставку на то, что другие выявленные риски окажут меньшее влияние на наше качество. Давайте рассмотрим, как выявить риски, определить их приоритеты и приступить к формированию стратегии.

3.2.1. Обучение выявлению рисков

Существует ряд методов выявления рисков, но в конечном итоге оно требует от нас просто скептического мышления. Такое мышление позволяет анализировать то, что мы знаем о контексте, и задавать себе вопросы, которые ставят под сомнение наши предположения. Например, возвращаясь к функции релевантного поиска из главы 1, мы можем задать следующие вопросы:

- Можем ли мы быть уверены в релевантности результатов поиска?
- Что делать, если поиск работает не так, как мы ожидали?
- Какие операторы поиска мы намерены поддерживать и почему?

Задавая подобные вопросы и рассматривая ответы на них, мы можем начать выявлять риски.

Скептицизм не означает «негативное или пессимистическое отношение» к той или иной ситуации. Тестирование иногда приобретает репутацию негативного отношения к работе других людей, но настоящее намерение состоит не в том, чтобы разрушить результаты чужого тяжелого труда. Тестирование необходимо для того, чтобы подвергнуть сомнению неизвестное в нашей работе и убедиться, что оно не скрывает неожиданных или нежелательных проблем. Будучи скептиками, мы думаем об истине, а не стремимся доказать, что что-то правильно или неправильно.

Когда мы учимся выявлять риски, нам нужно сосредоточиться на неизвестном и определить, какие риски скрываются в этих областях, которые, возможно, следует проверить. Это может показаться довольно абстрактным упражнением, но к счастью, есть несколько техник и инструментов, которые помогают сфокусировать внимание на некоторых областях работы.

3.2.2. Игра «Газетный заголовок»

Игра «Газетный заголовок» (The Headline game) — это подход к выявлению рисков, который был популяризирован Элизабет Хендриксон (Elisabeth

Hendrickson) в книге *«Explore It!»* (серия Pragmatic Bookshelf, 2013). С помощью этого подхода можно определить области для исследовательского тестирования (https://pragprog.com/titles/ehxta/explore-it/). Мы рассмотрим исследовательское тестирование позже, но технику игры в газетный заголовок стоит изучить сейчас. Этот инструмент эффективен для анализа и выявления рисков, независимо от того, какое тестирование вы хотите провести.

Техника работает следующим образом: нужно представить ряд выдуманных заголовков, а затем определить риски, которые могут способствовать появлению каждого из них. Например, мы можем придумать такой газетный заголовок для платформы restful-booker:

> Позор для местного бизнеса! Отель B&B переполнен гостями с двойными бронированиями

Отталкиваясь от этого заголовка, рассмотрим риски, которые могут привести к такому сценарию. Например:

- Функция проверки двойного бронирования не работает.
- Бронирования сохраняются некорректно.
- Сохраненные бронирования аннулируются без уведомления гостя.

Заголовки действуют как триггеры, позволяющие выявлять риски, которые мы хотели бы устранить. В сочетании с качественными характеристиками они являются отличным способом определять риски, которые имеют для нас высокое значение. Как мы вскоре узнаем, в технике RiskStorming мы можем использовать качественную характеристику в качестве триггера для нахождения релевантных заголовков, которые, в свою очередь, помогут определить типы рисков, имеющих наибольшее значение. Более подробное руководство по игре в газетные заголовки вы можете прочитать в статье *«The Nightmare Headline Game»* (http://mng.bz/J2M0).

3.2.3. Обходное тестирование

Игра «Газетный заголовок» требует хорошего воображения, чтобы придумать материал для старта. Не все обладают этой способностью. К счастью, те, кому трудно дается придумывание заголовков, могут применить обходное тестирование (Oblique testing) для начала анализа рисков. Подход основан на так называемой обходной стратегии (Oblique strategy), созданной Брайаном Ино (Brian Eno). Брайан Ино придумал специальные карты, чтобы помочь

художникам исследовать различные творческие направления. Майк Талкс (Mike Talks) создал подобные карты *Oblique testcards*, чтобы использовать обходную концепцию для исследования перспектив тестирования. Колода содержит 28 карт с фальшивыми отрицательными отзывами. Вы выбираете наугад одну из карт, и написанный на ней отзыв помогает появлению идей для тестирования. Например, там написано:

Я думал, что это упростит мне жизнь! ОДНА ЗВЕЗДА!!!

Мы можем использовать эту цитату в качестве триггера, чтобы подумать о том, в каких случаях наши продукты могут вызвать у пользователя желание написать такой отзыв. Для платформы restful-booker это может быть связано с качеством документации на API, обеспечением обратной связи, числом вызовов, которые нужно сделать, чтобы достичь чего-то, и т. д.

Эффективность карточек в выявлении рисков обусловлена тем, что отзывы на них и достаточно конкретные, чтобы дать нам отправную точку для выявления рисков, и довольно общие, чтобы при многократном использовании помогать генерировать разные идеи. Колоду карт для обходного тестирования можно заказать на сайте Leanpub (https://leanpub.com/obliquetesting).

3.2.4. Техника RiskStorming

Хотя игра «Газетный заголовок» и методы обходного тестирования полезны для инициирования идей, они требуют как минимум базового опыта и навыков в анализе рисков. Для команд, которые только начинают анализировать риски, подойдет техника RiskStorming.

RiskStorming основана на использовании колоды карт *TestSphere*, созданной Береном ван Дале (Beren Van Daele). Подробнее о TestSphere можно узнать на сайте Министерства тестирования (https://www.ministryoftesting.com/testsphere). Программа RiskStorming, разработанная в сотрудничестве с Марселем Геленом (Marcel Gehlen) и Андреа Йенсен (Andrea Jensen), состоит из трех этапов:

1. Определение характеристик качества, которые могут повлиять на продукт или конкретный параметр.
2. Выявление рисков, которые могут повлиять на характеристики качества.
3. Определение способа проверки рисков.

Подобно карточкам обходного тестирования, карты TestSphere помогают рождению идей и проведению дискуссий о качестве и стратегии тестирования.

Колода довольно большая — в ней 100 карт, разделенных на пять категорий, которые охватывают такие темы, как качество, методы тестирования и тестовые паттерны. RiskStorming направляет команду на выявление рисков, но не препятствует появлению оригинальных идей. Давайте попробуем с помощью RiskStorming выявить риски на платформе restful-booker, выполнив первые два этапа этой техники.

Этап 1. Качество

На первом этапе нам необходимо выбрать из колоды шесть из двадцати карт, относящихся к категории качества. Как мы обсуждали ранее в этой главе, характеристики качества определяются, исходя из нашего понимания требований пользователей. Соответственно, выбранные карты должны отражать эти требования. Пользователям платформы restful-booker, гостям и администраторам, требуются характеристики качества, определенные в разделе 3.1.3:

- интуитивность;
- завершенность;
- стабильность;
- конфиденциальность;
- доступность.

Выберем из колоды шесть карт с характеристиками качества, которые либо совпадают, либо похожи на наш список (со списком карт с характеристиками качества можно ознакомиться, попробовав бесплатную онлайн-версию игры RiskStorming на сайте https://app.riskstorming):

- удобство для пользователя;
- функциональность;
- доступность;
- стабильность;
- способность повысить ценность бизнеса;
- безопасность.

Этап 2. Риски

Теперь мы можем рассмотреть шесть характеристик качества и определить риски, которые могут потенциально на них повлиять. Например, мы можем

использовать игру «Газетный заголовок», чтобы придумать, как можно негативно повлиять на «удобство для пользователя». Мы можем добавить столько рисков, сколько захотим, но рекомендуется, чтобы этап был ограничен по времени, чтобы не дать спискам выявленных рисков слишком разрастись. Помните, что мы не можем протестировать всё.

Мы ограничим количество рисков тремя на каждую характеристику качества. Получится следующий список:

- удобство для пользователя:
  - при успешных запросах отправляются неправильные коды состояния или ответы;
  - неясные или неправильные сообщения об ошибках пользователя;
  - документация, описывающая запросы API, трудна для восприятия;
- функциональность:
  - существующие функции перестают работать при добавлении новых возможностей;
  - реализованные функции не соответствуют требованиям пользователей;
  - валидация на API не работает так, как ожидалось;
- доступность:
  - API перестают работать при обращении слишком большого количества пользователей;
  - утечки памяти приводят к сбоям в работе API;
  - невозможно бронирование номеров, обозначенных как свободные;
- стабильность:
  - API в платформе отправляют некорректные HTTP-запросы и ответы друг другу;
  - API не удается корректно развернуть;
  - периодические сбои из-за API;
- способность к повышению ценности бизнеса:
  - администратор не может обновить внешний вид и функциональность гостевых страниц;
  - введенные администратором данные не сохраняются и не передаются корректно;
  - функции не соответствуют требованиям администратора;

- безопасность и разрешение:
  - функциональные возможности, предназначенные только для администратора, доступны другим пользователям;
  - личные данные могут быть украдены;
  - сайтом можно управлять извне для атаки на пользователей.

Перечисленные риски определяют шаги по реализации стратегии тестирования. Мы можем дополнить нашу модель стратегии сокращенным списком рисков (я выбрал по одному из каждого раздела), как это показано на рис. 3.3.

Рис. 3.3. Окончательная модель стратегии, показывающая, как выявленные и изученные риски определяют шаги на пути к достижению целей

Мы перечислили лишь несколько приемов, которые помогут начать выявлять риски, но лучший способ совершенствования анализа рисков — это практика. Подобно регулярному пересмотру характеристик качества, мы также должны оценивать риски, которые могут повлиять на нашу работу. Если этот процесс регулярен, мы сможем не только сохранять список рисков в актуальном состоянии, но и будем совершенствовать свои навыки выявления рисков.

Упражнение

Чтобы проверить, как работает RiskStorming, зайдите на ее страницу (http://mng.bz/woeq) и загрузите PDF-файлы с упражнениями «Звезда смерти» (Death Star) или «Железный человек» (Iron Man). Попытайтесь выполнить первые два этапа сессии RiskStorming и посмотрите, какие риски вы придумали. Вы можете использовать бесплатную онлайн-версию RiskStorming (https://app.riskstormingonline.com/), чтобы помочь выявить риски в вашем проекте.

3.3. ПЕРВЫЕ ШАГИ В РЕАЛИЗАЦИИ СТРАТЕГИИ

Теперь, когда цели и риски определены, нам остается последнее: установить приоритеты, то есть то, на каких рисках мы хотим сосредоточиться и в каком порядке. Для этого мы можем использовать три критерия, первый из которых — характеристики качества. Например, в случае с платформой restful-booker мы указали интуитивность в качестве первой из них. Предположим, что это самая важная характеристика качества и мы хотим поставить на первое место связанные с ней риски. Это было бы разумно, но мы также должны учитывать два других критерия: вероятность и серьезность.

Насколько вероятно, что риск повлияет на качество? Если бы мы знали, что вероятность дождя составляет 80 %, то перед выходом на улицу, скорее всего, взяли бы зонтик. Если бы вероятность дождя была 20 %, то, возможно, обошлись бы без зонтика. Серьезность используется для оценки размера воздействия на качество. Эти два критерия используются вместе, чтобы помочь определить приориты рисков, как показано на рис. 3.4.

Если риск имеет высокую вероятность и серьезность, то его следует рассмотреть как приоритетный. А если вероятность и серьезность низкие, то риск может опуститься вниз в списке приоритетов. Эти критерии полезны в сочетании с характеристиками качества. Например, у нас есть два риска:

- Бронирование, сделанное в прошлом, отправляет непонятное сообщение об ошибке (интуитивность).
- Администратор не может добавить номера для бронирования (завершенность).

Если второй риск имеет высокую вероятность и серьезность, то имеет смысл отдать предпочтение именно ему. Это упрощенный пример, потому что вероятность и серьезность не измеряются двумя оценками, как просто «высокие» или

«низкие». Важно помнить, что определение приоритетов рисков — это в конечном итоге игра в угадайку. Мы должны сделать все возможное, чтобы наши решения о приоритетах рисков были как можно более обоснованными. В этом поможет регулярная переоценка как наших целей в области качества, так и связанных с ними рисков.

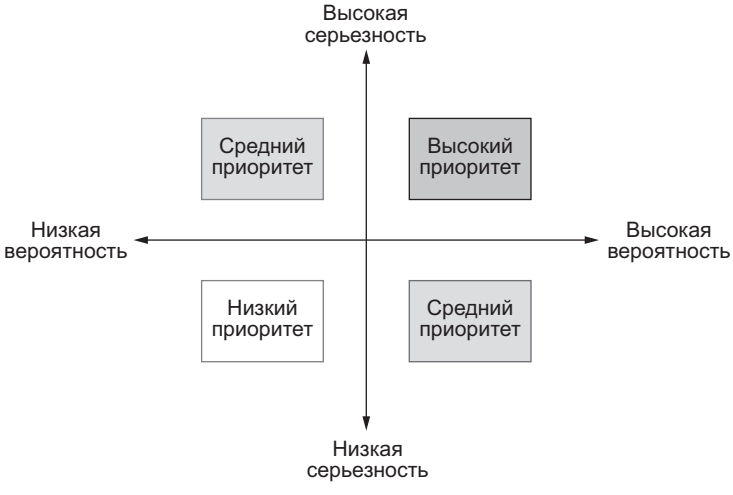

Рис. 3.4. Модель приоритизации рисков на основе их вероятности и серьезности

Упражнение

Рассмотрите риски, которые мы определили в предыдущем разделе, и, основываясь на знании характеристик качества и самого продукта, попытайтесь упорядочить их по приоритетам. Подумайте об оценках вероятности и серьезности. Доверяете ли вы своей интуиции или используете более формальный подход, оценивая их по трехбалльной шкале (высокие, средние или низкие)?

3.3.1. Выбор правильного подхода к тестированию риска

Наша стратегия начинает складываться. У нас есть цели и риски, на которых мы должны сосредоточиться. Теперь мы можем перейти к выбору типа тестирования. Допустим, что мы выбрали следующий риск:

Неясные или неправильные сообщения об ошибках пользователя.

Он может повлиять на интуитивность. Мы можем провести тестирование дизайна API, чтобы убедиться, что сообщения об ошибках, которые мы отправляем, не являются некорректными или неясными.

Это только один пример. В наших проектах будет много рисков, которые нужно устранять разными способами. Некоторые виды тестирования могут устранять несколько рисков. Но чтобы определить подход к тестированию каждого риска, необходимо иметь некоторое представление о возможностях тестирования. Успешная стратегия будет использовать возможности в диапазоне от применения средств автоматизации до тестирования абстрактных концепций и идей, которое мы рассмотрим в следующей части. Пока поставим создание стратегии на паузу, чтобы узнать о возможностях тестирования и типах рисков, которые они помогают устранить. Затем вернемся к нашей стратегии и определим, какое тестирование мы хотим провести и когда.

ИТОГИ

- Для успешной стратегии необходима цель, к которой нужно стремиться, иначе она может потерять смысл.
- Цель нашей стратегии — повышение качества.
- Каждый человек имеет собственное представление о качестве, которое имеет значение только в определенный момент времени. Поэтому мы должны понять, что означает качество для наших пользователей.
- Мы можем определить значение качества для наших пользователей различными способами с помощью характеристик качества.
- Значение качества для наших пользователей может использоваться как отправная точка для определения рисков.
- Определение рисков требует скептического мышления для выявления неизвестных, которые необходимо исследовать.
- Для выявления рисков мы можем использовать такие методы, как «Газетный заголовок», обходное тестирование и RiskStorming.
- Мы определяем приоритеты рисков на основе вероятности и серьезности.
- После получения списка рисков мы можем приступать к определению подхода к тестированию каждого риска.

Часть 2

Разработка стратегии тестирования

Когда речь заходит о стратегиях тестирования, бывает трудно отделить причины от следствий. До погружения в работу с конкретными операциями тестирования необходимо понять рабочий контекст. Но чтобы знать, какие действия выполнять в первую очередь, нужно иметь представление о том, как их выполнять. Поэтому, прежде чем приступить к формированию стратегии тестирования, давайте рассмотрим виды деятельности, которыми мы можем воспользоваться в течение всего жизненного цикла разработки программного обеспечения. Затем мы узнаем, как их интегрировать в нашу стратегию.

В главе 4 мы рассмотрим тестирование идей и дизайна веб-API. В главе 5 разберемся, каким образом исследовательское тестирование помогает понять, как на самом деле ведут себя веб-API. В главе 6 узнаем, как использовать автоматизацию для поддержки нашей стратегии тестирования API. Наконец, в завершение этой части обсудим, как сформировать стратегию, использующую изученные нами виды тестирования для решения конкретных задач.

4 Тестирование дизайна API

В этой главе

- ✓ Использование техники вопросов для проверки дизайна и идей
- ✓ Совместная работа команд над дизайном API, чтобы облегчить тестирование
- ✓ Документирование API для улучшения общего понимания и помощи в тестировании
- ✓ Место тестирования дизайна API в общей стратегии тестирования

Представьте, что кто-то изучает нашу «песочницу» — API платформы restful-booker, чтобы определить, подходит ли это приложение для отеля B&B. Этот пользователь выясняет, что для создания бронирования номера необходимо вызвать `room` API. В документации API описаны детали, подобные тем, что показаны на рис. 4.1.

На основе этой информации пользователь создает и отправляет следующий HTTP-запрос, чтобы посмотреть, как ответит API:

```
POST /room/ HTTP/1.1
Host: automationintesting.online
```

```
Accept: application/json
Cookie: token=r76BXGVy8rlASuZB

{
    "roomNumber":100,
    "type":"Single",
    "accessible":false,
    "description":"Введите описание номера",
    "image":"https://www.mwtestconsultancy.co.uk/img/room1.jpg",
    "roomPrice":"100",
    "features":["WiFi"]
}
```

CREATE ROOM

`/room/`

HTTP-метод	Описание
POST	Создает комнату (номер) для бронирования

Аргументы запроса

`roomNumber`	Номер комнаты (номера)
`type`	Тип номера (одноместный, двухместный, двухкомнатный, семейный, люкс)
`accessible`	Указывает, имеет ли комната доступ для людей с ограниченными возможностями
`description`	Задает описание комнаты
`image`	Задает URL-адрес изображения номера
`roomPrice`	Устанавливает стоимость номера
`features`	Особенности номера (Wi-Fi, телевизор, сейф, радио, прохладительные напитки, хороший вид из окна)

Рис. 4.1. Пример документации к API, которая может быть представлена на сайте разработчика

Однако вместо ожидаемого создания комнаты в HTTP-ответе приходит сообщение об ошибке 500 (`500 Server Error`). Эта ошибка оставляет пользователя в недоумении. Он возвращается к документации, чтобы посмотреть, что мог

упустить, но не узнает ничего нового. В конце концов он решает, что API работает неправильно, и уходит, в результате чего мы теряем клиента.

Что же пошло не так? На первый взгляд кажется, что `room` API не работает. Но давайте рассмотрим процесс принятия решения о создании конечной точки `POST /room`. С нашей точки зрения, API работает правильно, создан в соответствии с предоставленными требованиями и развернут без проблем. С точки зрения наших ожиданий все в порядке. Но именно здесь и возникает проблема различия ожиданий. Когда наша команда начала разрабатывать API, она выбрала следующую концепцию:

> Идентификатор номера всегда должен иметь тип string, поскольку некоторые пользователи используют нецелые числа для идентификации номеров.

Но команда упустила из внимания ряд вопросов. Что, если кто-то отправит целое число? Как мы должны его обработать? Если бы команда задалась этим вопросом, то могла бы прийти к решению отправлять ответ `400 Bad RequestHTTP` с дополнительной информацией, чтобы сообщить клиенту, что требуется именно строка. Дальнейшее обсуждение могло бы выявить, что на самом деле `roomNumber` — это вводящее в заблуждение имя параметра и что переименование его в `roomName` прояснило бы ситуацию. Дополнительные обсуждения и вопросы при разработке могут играть решающую роль в том, решит ли потом человек, оценивающий платформу, использовать ее.

Этот пример демонстрирует, что решения, которые мы принимаем, и предположения, вытекающие из этих решений, влияют на качество создаваемых продуктов. Чаще всего проблемы, возникающие в наших системах, появляются не из-за неправильного кода, а из-за следующих рисков:

- Неправильная интерпретация требований пользователя.
- Неверные предположения.
- Нарушение принципов архитектуры.

Игнорирование этих рисков может привести к созданию продукта, который ведет себя не так, как ожидает или хочет пользователь. Это негативно скажется на развитии команды или приведет к появлению неожиданных ошибок и уязвимостей в нашем программном обеспечении. Поэтому важно обеспечить четкое и одинаковое понимание всеми членами команды того, что должно быть создано. В этой главе мы узнаем, как использовать методы тестирования, чтобы проверять проекты и идеи, устранять недопонимание, обнаруживать потенциальные проблемы и способствовать повышению качества продукта.

4.1. КАК МЫ ТЕСТИРУЕМ ДИЗАЙН API

В отличие от тестирования приложения, при тестировании дизайна API проверяются менее осязаемые вещи, такие как идеи, документация и визуализация. Перед созданием кода команда в идеале должна совместно обсудить проблему или требуемую функциональность и в результате выработать некий дизайн или соглашение о том, что нужно сделать. Именно во время этих обсуждений мы можем слушать, учиться и задавать вопросы, чтобы помочь командам установить общее понимание и выявить проблемы на более ранней стадии.

4.1.1. Инструменты для опроса

Основная проблема при тестировании API-проектов — научиться задавать правильные вопросы в нужное время. Конечно, для этого требуется опыт, но у нас есть методы, которые помогут начать задавать вопросы прямо сейчас.

Одним из распространенных приемов является техника шести ключевых слов, которые могут вдохновить нас на различные типы вопросов:

- кто;
- что;
- где;
- когда;
- почему;
- как.

Выберем один из шести пунктов списка, например «что», и используем его в качестве триггера, чтобы посмотреть, какие вопросы вы можете придумать. Например:

- Что происходит, когда пользователь не авторизован?
- Что будем делать, если API, от которого мы зависим, окажется недоступным?
- Что мы будем отправлять?

Порядок, в котором перечислены ключевые слова, случаен и не указывает на их приоритет. Каждое слово в списке позволяет исследовать идеи различными способами. Применяя такую технику, мы можем быстро подготовить вопросы.

Используйте навыки критического мышления, чтобы копнуть глубже, и навыки нестандартного мышления, чтобы исследовать различные идеи.

Чтобы помочь лучше понять силу вопросов, давайте начнем тестировать дизайн API, который сосредоточен на конечной точке `/room/`, являющейся частью `room` API в нашей песочнице. Допустим, один из членов нашей команды представил следующую историю пользователя:

> Для того чтобы можно было забронировать номер как владелец отеля B&B, я хочу иметь возможность создать номер.

Команда предложила следующую реализацию запросов и ответов для `room` API:

Запрос

```
POST /room/ HTTP/1.1
Host: example.com
Accept: application/json
Cookie: token=abc123

{
    "roomName": "102",
    "type": "Double",
    "accessible": "true",
    "description": "Это описание номера",
    "image": "/img/room1.jpg",
    "roomPrice": 200,
    "features": ["TV", "Safe"]
}
```

Ответ

```
HTTP/1.1 201 OK
Content-Type: application/json

{
    "roomid": 3,
    "roomName": 102,
    "type": "Double",
    "accessible": true,
    "image": "/img/room1.jpg",
    "description": "Это описание номера",
    "features": ["TV", "Safe"],
    "roomPrice": 200
}
```

Имея в своем распоряжении эту информацию, давайте применим технику шести вопросов, чтобы выявить потенциальные проблемы.

Кто?

Вопросы *кто?* полезны для получения дополнительной информации о людях или системах, взаимодействующих с нашими API. Чем больше мы узнаем о том, кто использует наши API, тем лучше сможем понять, каковы их потребности и как мы можем предоставить им больше возможностей. Примеры вопросов:

- *Кто будет использовать это?* Этот вопрос помогает лучше понять, кто или что будет использовать конечную точку `/room/`. Мы можем обнаружить, что пользователем будет другой API, библиотека пользовательского интерфейса или человек. То, что мы узнаем, может послужить основой для других вопросов. Например, если `/room/` использует другой API, мы можем задать технические или архитектурные вопросы.
- *Кто должен иметь доступ?* Мы можем рассмотреть и риски, связанные с безопасностью. В дизайне API упоминается заголовок `Cookie` с маркером, и видимо, надо узнать больше об элементах управления безопасностью, связанных с этим маркером. Хотя это довольно поверхностный вопрос по безопасности API (мы рассмотрим более подробно тестирование безопасности в главе 9), ответы на него могут позволить глубже изучить защиту наших API и исследовать риски, которые могут привести к уязвимостям.

Что?

Вопросы *что?* можно использовать для гипотетических ситуаций: *что, если что-то случится?* Эти вопросы требуют нестандартного мышления, предлагая сценарии, которые мы, возможно, еще не рассматривали. Полезный способ использовать вопросы *что?* — это взять уже имеющуюся информацию и попытаться мысленно продвинуться немного дальше. Примеры таких вопросов:

- *Что делать, если мы отправили неверные данные? Что произойдет?* Эти вопросы напоминают начальный пример, который мы рассматривали в этой главе. Не все гипотетические вопросы приводят к положительному результату, и нужно помнить о том, что делать с отрицательными результатами. Как минимум можно обсудить, как минимизировать ущерб, который может быть нанесен нашим системам. Но учитывая, что мы работаем с API, также стоит обсудить, как понятным образом обеспечить обратную связь при ошибках.
- *Что представляет собой полезная нагрузка и почему?* Здесь мы снова сосредоточены на том, чтобы узнать больше о принятых технических решениях. Сам по себе вопрос «Что представляет собой полезная нагрузка?» может

привести к простому ответу: JSON. Но добавив в конце вопрос *почему?*, мы просим дать более глубокое объяснение. Мы можем обнаружить, что существует несколько вариантов того, кто может использовать ответ API.

Где?

Вопросы *где?* могут быть использованы для расширения нашего представления о дизайне. С их помощью можно выявить потенциальные зависимости и узнать больше о том, какое влияние окажет выбранная архитектура на потенциальных потребителей ответов. Например, вы можете задать следующие вопросы:

- *Где будет сохранена комната?* Подобный вопрос поможет выявить детали реализации дизайна. Он инициирует дискуссию о том, как выбор реализации согласуется с требованиями архитектуры, которые необходимо выполнить. Например, если бы мы работали в микросервисной архитектуре, означало бы это необходимость создания нового API или нет?
- *Где будет использоваться комната?* Этот вопрос несколько отличается от вопроса «Кто будет использовать это?». Он помогает разобраться, как дизайн согласуется со сферой применения платформы или пользовательских функций. Изучение области использования может помочь выявить новых пользователей — людей или API.

Когда?

Вопросы *когда?* позволяют сфокусироваться на времени. Они помогут определить события, которые могут произойти в будущем, а также то, что произойдет во время этих событий. Примеры вопросов о времени:

- *Когда мы будем вносить изменения в контракт API, будем ли мы изменять URI?* Когда мы думаем о решении конкретной проблемы, важно учитывать его долгосрочное будущее. Как оно может измениться со временем и как часто мы будем вносить изменения? Что может вызвать изменения? Как мы будем обрабатывать их? Вопрос об управлении версиями будет иметь последствия не только для дизайна API, но и для того, как мы документируем изменения и информируем наших пользователей.
- *Когда это заработает, хотим ли мы кэшировать запросы?* Особенно интересна первая часть этого вопроса, затрагивающая проблемы выпуска и поведения наших производственных сред. Она поможет убедиться, что дизайн работает в соответствии с правилами и шаблонами наших платформ. Положительный ответ о кэшировании затрагивает разработку (например, надо убедиться, что

в URI не используются произвольные переменные, которые могут нарушить правила кэширования).

Почему?

Вопросы *почему?* позволяют глубже проникнуть в основу и смысл принимаемых решений. Они помогут осознать полезность тех или иных действий и объяснить другим процесс принятия решений. Примеры вопросов:

- *Почему мы это создаем?* Возможно, немного спорный, но все же важный вопрос. Если мы собираемся вкладывать свое время и деньги в создание функций, лучше, чтобы мы знали, в чем польза этой работы. Подобный вопрос поможет понять конечного пользователя и войти в его положение. Ответы предоставят больше сведений о проблемах, с которыми может столкнуться пользователь, и мы сумеем сделать обоснованные выводы о том, как мы будем решать эти проблемы.
- *Почему мы решили организовать функции в виде массива?* Этот вид вопросов фокусирует больше внимания на выборе дизайна. Почему функции представлены в виде массива? Если это фиксированный список функций для выбора, почему бы не иметь параметры для подобъектов? Цель не в том, чтобы критиковать принятые решения, а в том, чтобы изучить альтернативы и понять сделанный выбор.

Как?

Вопросы *как?* помогают нам понять, как что-то будет работать или какие задачи мы должны решить для успешного выполнения нашей работы. Примеры вопросов:

- *Как это будет работать?* Когда вы задаете вопросы в контексте совместной работы, может возникнуть ощущение, что они не имеют смысла. «Как это будет работать?» является отличным примером, потому что на первый взгляд кажется, что вы открываете себя для критики за непонимание обсуждаемого проекта. Однако подобный вопрос может выявить неверные толкования одних и тех же вещей внутри команды. Я задавал этот вопрос, если один из членов команды предлагал спорное решение. Иногда базовые вопросы работают лучше всего.
- *Как мы будем это тестировать?* Не все вопросы должны быть связаны с основами проекта. Мы также можем спросить, как мы будем реагировать на принятые решения. Такой вопрос позволяет выявить потенциальные помехи

тестированию. Например, если мы разработали что-то, срабатывающее через определенные промежутки времени, то следует спросить, как мы будем это тестировать. В этом случае требуется контроль над временны́ми интервалами, чтобы ускорить тестирование.

Упражнение

Используя технику шести вопросов, просмотрите еще раз конечную точку `POST /room/` и запишите список вопросов, которые вы могли бы задать команде, чтобы узнать дополнительную информацию. Постарайтесь записать не менее трех вопросов для каждого пункта.

Проверять идеи — значит задавать вопросы

Такие приемы, как техника шести вопросов, являются отличным стартом. Но для дальнейшего развития навыков постановки вопросов стоит понимать, что важную роль в тестировании играет критическое и нестандартное мышление.

Давайте вернемся к нашей модели тестирования, показанной на рис. 4.2. Подобно тому как при тестировании приложения мы можем задать себе вопрос «А что, если я тысячу раз нажму эту кнопку "Добавить"?», мы можем задать вопрос типа «Кто будет использовать этот ответ API?».

Рис. 4.2. Демонстрация того, как могут при тестировании различаться вопросы, задаваемые в воображении и в области реализации

Мы можем фокусироваться на разных вещах, например на программном обеспечении или идеях, но нашу способность задавать полезные вопросы определяет критическое и нестандартное мышление. Обладая навыками критического мышления, мы можем задавать вопросы, которые позволяют докопаться до истины и смысла идей, выявить мотивы принимаемых решений и риски, которые могут привести к ошибкам. Обладая навыками нестандартного мышления, мы можем придумать вопросы, которые помогут расширить понимание последствий принятых решений. Грубо говоря, критическое мышление позволяет нам копать глубже, а нестандартное — шире.

Оттачивание критического и нестандартного мышления помогает придумывать более действенные вопросы, хотя их совершенствование может показаться абстрактным. Лучший способ улучшить оба навыка — думать о них как о мышцах. Если находить возможности для тренировки обоих, тестируя идеи и код, это сделает ваши навыки тестирования более эффективными.

Вопросы, которые мы рассмотрели, являются лишь небольшой демонстрационной выборкой. Существует множество вопросов, чтобы узнать больше о том, какой выбор мы сделали для дизайна API, как он может повлиять на пользователей и что может произойти в конкретных ситуациях. Однако мы неизбежно придем к тому моменту, когда источник вопросов иссякнет. Это может быть явным признаком того, что мы исчерпали все возможные пути и пора прекращать тестирование. Однако мы можем использовать некоторые дополнительные техники и инструменты, которые помогут нам генерировать другие идеи.

4.1.2. Расширение методов и инструментов тестирования дизайна API

Техника шести вопросов — это основа для тестирования дизайна API, но она может быть дополнена с помощью других методов опроса, анализа типов данных и визуализации. Каждая из этих техник позволяет генерировать новые идеи, которые можно использовать в сочетании с шестью вопросами для дальнейшего и более глубокого тестирования.

Дополнительные методы опроса

Для расширения списка вопросов вы можете использовать две техники: слово *«еще»* и метод воронки.

Добавление слова «еще» после одного из вопросительных слов побуждает нас и команду мыслить нестандартно. Примеры:

- Кто еще собирается использовать это?
- Что еще может произойти?
- Как еще может возникнуть ошибка?

Если вопросы с «еще» помогают нам мыслить более широко, то метод воронки стимулирует критическое мышление. В отличие от техник, использующих ключевые слова для создания вопросов, метод воронки подразумевает реакцию на ответы с последующими вопросами для более глубокого изучения проблемы. Пример:

В: Как пользователь добавляет изображение?

О: Пользователь добавляет изображение, предоставляя URL-ссылку в формате строки.

В: Почему пользователь должен предоставить URL изображения, а не загрузить его?

О: Потому что у нас пока нет функции загрузки.

В: Когда мы ее создадим?

О: Возможно, нам стоит сделать это прямо сейчас.

В этом примере цель не в том, чтобы выставить отвечающего в плохом свете, а в том, чтобы докопаться до причин принятого решения. Чтобы добиться успеха при этом подходе, мы должны использовать другие навыки постановки вопросов. В частности, необходимо использовать навыки активного слушания, чтобы реагировать на ответы. Кроме того, необходимо следить за формулировками вопросов, чтобы они демонстрировали проницательность, но не агрессию.

> Если вам трудно запомнить техники постановки вопросов или вы хотите для разнообразия использовать случайный подход, то можете бесплатно скачать карточки Discovery Cards от Hindsight Software (https://behavepro.app/agile-coaching-tools/discovery-cards). Их можно использовать в качестве подсказок при постановке вопросов.

Упражнение

Попробуйте расширить первоначальные вопросы из предыдущего задания, используя слово «еще». Подумайте, как меняются вопросы. Запишите каждый из вопросов, которые вы могли бы задать.

Анализ типов данных

Анализ типов данных в нашем дизайне API — важная задача при определении стратегии тестирования. В качестве примера напомним полезную нагрузку при создании комнаты:

```
}
    "roomName": "102",
    "type": "Double",
    "accessible": "true",
    "description": "Это описание номера",
    "image": "/img/room1.jpg",
    "roomPrice": 200,
    "features": ["TV", "Safe"]
}
```

Мы отправляем `RoomPrice` в виде целого числа. С учетом этой информации мы можем задать следующие вопросы, связанные с типом данных:

- Что, если вместо целого числа (integer) мы отправим число с плавающей точкой (float)?
- Что, если мы пошлем слишком большое число?

Также мы можем задать вопросы, связанные с бизнес-правилами, лежащими в основе выбора типа данных:

- Что делать, если значение `RoomPrice` меньше или равно нулю?
- Что делать, если `RoomPrice` вообще не отправлен?

Зная, как работает тип данных integer, а также бизнес-правила, мы можем проверять конкретные детали.

У каждого типа данных есть свои особенности, и необходимо понимать принципы их работы и ограничения. К счастью, существуют шпаргалки. Например, мы

можем воспользоваться шпаргалкой *«Test Heuristic Cheat Sheet»* (https://goritskov.com/media/files/testheuristicscheatsheetv1.pdf), созданной Элизабет Хендриксон (Elisabeth Hendrickson). Мы обсудим эвристики более подробно в главе 5, когда будем говорить об исследовательском тестировании, а пока вы можете использовать их как средство, помогающее быстро задавать вопросы.

Упражнение

Используя шпаргалку Test Heuristic Cheat Sheet, напишите список вопросов о полезной нагрузке `POST /room/`.

Визуализации

Как ни банально это звучит, но картинка стоит тысячи слов. Когда мы обсуждаем какую-либо информацию, визуализация может быть чрезвычайно ценной.

Как это будет работать?

Представьте, что вместо устного ответа член команды подошел к доске и нарисовал диаграмму, показанную на рис. 4.3.

Рис. 4.3. Пример визуализации для дизайна API, чтобы облегчить обсуждение

Хотя это не самая сложная визуализация, она представляет зависимости между API и демонстрирует механизм безопасности. Она также может послужить отправной точкой для постановки дополнительных вопросов:

- Что делать, если подключение к `Auth` API не удалось?
- Что делать, если `Auth` API возвращает код ошибки?
- Как `Auth` API сообщает `room` API, прошла ли проверка безопасности?

Визуализация служит основой для постановки новых вопросов, а также инструментом объяснения. Предложив собственные визуализации, можно получить ценную обратную связь от членов команды, которые сообщат, чем отличается их собственное понимание того, что должно быть построено.

Упражнение

Используя свои знания о платформе restful-booker и модель, которую вы построили в главе 2, попробуйте создать упрощенную визуализацию процесса `POST /booking/`. Добавьте в свой список дополнительные вопросы, которые вы могли бы задать, глядя на эту визуализацию.

Как уже говорилось ранее, тестирование дизайна API, абстрактных идей и конкретных требований — это навык, который приходит с практикой. Но методы и инструменты, которые мы рассмотрели, помогут сделать первые шаги и сформулировать вопросы. Однако есть еще один подход, который помогает начать обсуждение и повысить уровень общего понимания. Это изучение документации.

4.2. ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТОВ ДОКУМЕНТИРОВАНИЯ API ДЛЯ ТЕСТИРОВАНИЯ ДИЗАЙНА

Бытует мнение, что подготовка документации — это пустая трата ресурсов, поскольку документация редко используется, но при этом ее необходимо поддерживать. Примером служат спецификации объемом в сотни страниц. Однако если документация сделана правильно, она характеризуется сочетанием

легкости понимания и простоты поддержки. При этом в ней фиксируется то, что мы намереваемся создать. Документация может демонстрировать консенсус в команде и быть эффективным средством информирования людей, не входящих в команду.

Когда мы тестируем дизайн API, мы не просто раскрываем информацию и обнаруживаем проблемы. Мы помогаем командам зафиксировать общее понимание проблем и решить следующие задачи:

- устранение неоднозначностей и двусмысленностей, четкое указание, как будет работать наш API;
- определение новой информации, которую мы можем использовать для дальнейших исследований;
- четкое изложение деталей работы API для пользователей, будь то другая команда, разрабатывающая API на той же платформе, или сторонний человек.

Важно создать документацию, которая была бы полезной для всех и простой в обслуживании. К счастью, как мы узнаем далее, современные инструменты документирования разработаны с учетом этих требований.

4.2.1. Документирование API с помощью Swagger/OpenAPI 3

Существует множество инструментов для документирования API с различными функциями, но для нашей документации мы будем использовать Swagger. Этот набор инструментов мы обсудим в ближайшее время, а сейчас сосредоточимся на использовании спецификации OpenAPI 3 для документирования дизайна API.

OpenAPI 3 появилась как часть Swagger. Пытаясь помочь разработке промышленного стандарта документирования API, команда Swagger открыла исходный код и передала схему сообществу OpenAPI Initiative (https://www.openapis.org/). С тех пор OpenAPI Initiative превратила оригинальную схему Swagger в спецификацию OpenAPI 3, которая может использоваться для описания API в открытом и легко проверяемом виде, помогая снизить риски, связанные с неправильным толкованием проектов и документации.

Чтобы понять, как работает OpenAPI 3, давайте создадим проект API и используем схему для `POST room` API. Напомним вид запросов и ответов:

Запрос

```
POST /room/ HTTP/1.1
Host: example.com
Accept: application/json
Cookie: token=abc123
```

```
{
    "roomName": "102",
    "type": "Double",
    "accessible": "true",
    "description": "Это описание номера",
    "image": "/img/room1.jpg",
    "roomPrice": 200,
    "features": ["TV", "Safe"]
}
```

Ответ

```
HTTP/1.1 201 OK
Content-Type: application/json
```

```
{
    "roomid": 3,
    "roomName": 102,
    "type": "Double",
    "accessible": true,
    "image": "/img/room1.jpg",
    "description": "Это описание номера",
    "features": ["TV", "Safe"],
    "roomPrice": 200
}
```

Предпочтительным форматом OpenAPI является YAML. В зависимости от выбранного редактора или IDE, вы можете установить плагины или линтеры, чтобы документация соответствовала шаблонам OpenAPI. В качестве альтернативы, если вам нужны дополнительные возможности, существует SwaggerHub, который предлагает бесплатный для регистрации и использования онлайн-редактор (https://swagger.io/tools/swaggerhub/), хотя его продвинутые функции требуют платной подписки. Кроме того, если вам удобно использовать Docker, вы можете получить Docker-образ редактора. Подробности о Docker-образе можно найти на странице Swagger в GitHub (http://mng.bz/7yx9).

Альтернативные инструменты проектирования API

Если по какой-либо причине Swagger вам не подходит, существуют другие инструменты и средства проектирования API, поддерживающие формат OpenAPI, например Postman API design и Stoplight.

Независимо от выбранного инструментария, давайте начнем с создания нового YAML-документа проекта и добавим в него следующее:

```
openapi: 3.0.0
info:
  description: An example API design
  version: 1.0.0
  title: SandBox Room service
  contact:
    name: Mark Winteringham
```

В первой строке мы объявляем тип схемы для документации. Это необходимо для проверки формата нашего проекта на соответствие OpenAPI 3 и совместимости с другими инструментами Swagger. Далее мы предоставляем контекстную информацию в разделе `info`, описывающую API и версию проекта. С этими данными мы можем документировать запрос (request) и ответ (response), добавив в YAML-файл следующие данные:

```
servers:
  - url: https://example.com/
paths:
  /room/:
    post:
      tags:
        - room
      parameters:
        - in: cookie
          name: token
          required: true
          schema:
            type: string
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Room'
        description: roomPayload
        required: true
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Room'
        ‘403’:
          description: Forbidden
```

Как мы узнаем в следующем разделе, проектная документация может быть использована одним из инструментов Swagger для создания интерактивной

документации API. Это позволяет отправлять запросы и получать ответы. Чтобы включить эту функцию, нужно указать одну или несколько записей в разделе `servers` для документации, которую мы будем использовать.

Затем мы описываем конечную точку для `room` API, начиная со спецификации URI как `/room/` со следующими четырьмя разделами:

- *`tags`* — теги, позволяющие дать нашей спецификации URI имя, чтобы ее было легко найти в документации.
- *`parameters`* — в проекте мы можем указать заголовки, которые пойдут в запрос. В нашем запросе нам требуется заголовок `Cookie` со значением `token=abc123`.
- *`requestBody`* — хотя мы еще не указали, каким будет тело запроса, его нужно будет объявить. Раздел `content:` сообщает, что тело запроса будет в формате `application/json`, до того как будет создана ссылка на схему.
- *`responses`* — раздел, который позволяет перечислить ожидаемые ответы с кодами состояния, объявив каждый код подразделом `response`, и добавить дополнительные детали. Например, раздел `201` содержит дополнительные сведения, показывающие, что в ответ добавляется полезная нагрузка, соответствующая схеме `room`.

В последних двух разделах мы ссылались на схему `room` с помощью `$ref:'#/components/schemas/Room'`, поэтому давайте завершим наш YAML-файл, создав схему для следующего тела JSON:

```
{
    "roomid" : 1
    "roomName" : "101",
    "type": "Single",
    "accessible" : false,
    "description" : "Описание комнаты",
    "image" : "link/to/image.jpg",
    "roomPrice" : "100",
    "features" : ["TV", "Refreshments", "Views"]
}
```

Добавим следующее:

```
components:
  schemas:
    Room:
      title: Room
      type: object
      properties:
        accessible:
          type: boolean
```

```
      description:
        type: string
      features:
        type: array
      items:
        type: string
        pattern: Single|Double|Twin|Family|Suite
    image:
      type: string
    roomName:
      type: string
    roomPrice:
      type: integer
      format: int32
      minimum: 0
      maximum: 999
      exclusiveMinimum: true
      exclusiveMaximum: false
    roomid:
      type: integer
      format: int32
  required:
    - accessible
    - description
    - features
    - image
    - roomName
    - roomPrice
```

Обратите внимание, что первые три строки этого раздела имеют ту же иерархию, что и ссылка `$ref:'#/components/schemas/Room'`. Так мы связываем компонент `schemas` с разделом `paths`.

В разделе `Room` мы объявили структуру схемы, начиная с корневого объекта и его имени, затем следует раздел `properties`, где мы объявляем каждый элемент, который хотим видеть в объекте. Каждое свойство имеет правила для определения типа (строка, массив, целое число и т. д.), а также ограничивающие параметры.

Например, для `RoomPrice` мы используем параметры `minimum` и `maximum`, чтобы задавать минимальные и максимальные значения. Параметры `exclusiveMinimum` и `exclusiveMaximum` говорят, включаем ли мы установленное значение в допустимый диапазон. Так, например, минимальное значение установлено равным `0`, а значение `exclusiveMinimum` равно `true`, поэтому все, что меньше `1`, выходит за границы диапазона. Наконец, с помощью поля `required` мы можем установить, какие свойства являются обязательными, а какие — необязательными.

Сложив все это вместе, мы получаем документ проекта API для конечной точки `/room/`, который выглядит следующим образом:

```
openapi: 3.0.0
info:
  description: An example API design
  version: 1.0.0
  title: SandBox Room service
  contact:
    name: Mark Winteringham
servers:
  - url: example.com
servers:
  - url: https://example.com/
  paths:
    /room/:
      post:
        tags:
          - room
        parameters:
          - in: cookie
            name: token
            required: true
            schema:
              type: string
        requestBody:
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Room'
          description: roomPayload
          required: true
        responses:
          '201':
            description: Created
            content:
              application/json:
                schema:
                  $ref: '#/components/schemas/Room'
            '403':
              description: Forbidden
  components:
    schemas:
      Room:
        title: Room
        type: object
        properties:
          accessible:
            type: boolean
          description:
            type: string
```

```
      features:
        type: array
        items:
          type: string
          pattern: Single|Double|Twin|Family|Suite
      image:
        type: string
      roomName:
        type: string
      roomPrice:
        type: integer
        format: int32
        minimum: 0
        maximum: 999
        exclusiveMinimum: true
        exclusiveMaximum: false
      roomid:
        type: integer
        format: int32
  required:
    - accessible
    - description
    - features
    - image
    - roomName
    - roomPrice
```

Используя OpenAPI, мы можем быстро разработать документацию, которая легко читается и помогает в тестировании. Список свойств `properties` с полями `type` позволяет провести анализ типов данных, о котором мы говорили ранее. В разделе ответов `responses` можно проверить информацию об ошибках и обратной связи, которую мы отправляем. Теперь идея проекта больше не находится исключительно в голове одного или нескольких членов команды, а документирована для рассмотрения всей командой.

Упражнение

Использовать OpenAPI относительно просто, но требует некоторого знакомства с правилами схемы. Чтобы освоить документирование с помощью OpenAPI, создайте документ дизайна API для конечной точки `/booking`, которую мы тестировали ранее:

Запрос

```
POST /booking/ HTTP/1.1
Host: example.com
Accept: application/json
```

```
{
    "bookingdates": {
        "checkin": "2021-02-01",
        "checkout": "2021-02-03"
    },
    "depositpaid": false,
    "firstname": "Mark",
    "lastname": "Winteringham",
    "roomid": 1,
    "email": "mark@example.com",
    "phone": "01234567890"
}
```

Ответ

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "bookingid": 2,
    "booking": {
        "bookingid": 2,
        "roomid": 1,
        "firstname": "Mark",
        "lastname": "Winteringham",
        "bookingdates": {
            "checkin": "2021-02-01",
            "checkout": "2021-02-03"
        }
    }
}depositpaid": false,
```

4.2.2. Помимо документации

Одним из преимуществ использования OpenAPI являются широкие дополнительные возможности после завершения работы над документацией. Так, мы можем использовать инструменты Swagger, чтобы ускорить разработку и улучшить публичную документацию.

Swagger Codegen

Swagger Codegen позволяет превратить документацию по проектированию API в код. Это не только экономит время на создание нового API, но и помогает реализации текущего дизайна, оставляя свободу для добавления необходимой бизнес-логики. Конечно, если вы доверяете создание производственного кода стороннему инструменту, возникают дополнительные риски, но их можно

проверить с помощью других видов тестирования, которые мы рассмотрим в этой книге.

Подробнее о Swagger Codegen: https://github.com/swagger-api/swagger-codegen.

Swagger UI

Если вы использовали SwaggerHub, то, наверное, уже обратили внимание на интерактивную документацию справа от вашего проектного файла. Swagger UI способен представить документацию на дизайн в понятном пользовательском интерфейсе, который наглядно демонстрирует функциональность вашего API. Кроме того, есть возможность создавать запросы к вашему API на основе документации и отправлять их, а также просматривать ответы. Этот инструмент удобен для тестирования API, но что еще более важно, позволяет поделиться с пользователями точной документацией, показывающей, что может и чего не может ваш API. Благодаря этому устраняются риски, связанные с непониманием пользователями возможностей API.

Подробнее о Swagger UI: https://github.com/swagger-api/swagger-ui/.

> **Документация API GraphQL**
>
> GraphQL не использует OpenAPI и не поддерживает набор инструментов Swagger (на момент подготовки книги). Однако существуют аналогичные продукты с открытым исходным кодом, такие как graphql-playground и graphiql, которые поддерживаются сообществом GraphQL (https://github.com/graphql/).

4.3. СТИМУЛИРОВАНИЕ КОМАНДЫ К ТЕСТИРОВАНИЮ ДИЗАЙНА API

Знания и навыки, необходимые для анализа и проверки дизайна API, очень важны. Но без тестирования трудно помочь команде повысить качество. Да, мы можем тестировать проекты API асинхронно, тратя время на то, чтобы просмотреть дизайн, записать вопросы, а затем отправить их команде. Но такой подход отнимает много времени, а проблемы выявляются уже после начала внедрения, что сводит на нет пользу от тестирования. Кроме того, отсутствие совместного обсуждения минимизирует возможности для выявления несоответствий. Обмен электронными письмами или сообщениями в мессенджерах никогда не будет столь же эффективным, как обстоятельное обсуждение.

Чтобы получить максимальную отдачу от тестирования, его лучше всего основывать на дискуссии в группе. Присутствие разных сторон и направлений обеспечит различные точки зрения и источники информации, которыми можно воспользоваться. Задавая вопрос о продукте, можно вызвать обсуждение архитектурных ограничений. Получение разъяснений по конкретному вопросу может выявить различия в понимании в команде.

4.3.1. Как получить возможность тестирования дизайна API

Проведение обсуждений во многом зависит от особенностей мышления членов команды. Легко сказать, что вы должны использовать обычные регулярные встречи или сами организовывать совместные обсуждения. Но нередко команды проводят обсуждения в узком кругу, не приглашая тестировщиков. Конечно, можно попробовать напроситься, но в этом случае у вас не будет гарантии поддержки всей команды.

Как мы обсудим в последующих главах, для того чтобы побудить команду подключить тестирование уже на этапе разработки идей, необходима стратегия. Начните со встреч тет-а-тет с единомышленниками. Поделитесь с командой информацией о результатах таких встреч и, возможно, расширьте число их участников, чтобы привлечь больше внимания. Когда наберется критическая масса интереса, обсудите со всей командой формализацию такого тестирования. Или, если все и так хорошо, просто продолжайте выполнять свою работу.

4.3.2. Использование регулярных встреч

К счастью, тестирование дизайна API хорошо вписывается в подходы к командному обсуждению проектов. Независимо от стиля проведения обсуждения, тестирование можно использовать, чтобы прояснять идеи и глубже продумывать то, что мы создаем.

Формальные подходы

Agile-церемонии стали неотъемлемой частью цикла командной разработки программного обеспечения. Команды проводят регулярные встречи, на которых обсуждают предстоящую работу. Такие церемонии, как планирование спринта или kick-off-команды (определение ролей, правил работы, расписания встреч, критериев готовности), позволяют начать тестирование проекта API. Как

правило, на таких церемониях присутствуют люди с разными ролями, которые совместно обсуждают идеи, что равнозначно тестированию. Привлекая внимание к необходимости тестирования, вы сможете расширить круг обсуждаемых проблем.

Если вы хотите воспользоваться преимуществами регулярных церемоний, имейте в виду ограничения. Некоторые встречи могут быть довольно продолжительными, поскольку приходится укладывать обсуждение большого количества вопросов в отведенное время. Это может привести к усталости участников и затруднит общение с ними. Кроме того, в больших группах некоторые участники будут менее охотно делиться опытом.

Неформальные подходы

Хотя во время официальных церемоний можно прийти к нешаблонным решениям, неформальные встречи иногда будут более успешными.

Например, обсуждение дизайна только с одним коллегой может быть эффективным для тестирования. Можно утверждать, что работа в паре сама по себе подразумевает тестирование в неявном виде, поскольку происходит обсуждение идей и обмен решениями. Однако если пара сосредоточена на реализации, возможно предвзятое отношение к текущей задаче. Договорившись обсудить и протестировать идеи до их реализации, мы позволяем себе более широко и глубоко обдумать будущую работу.

Помимо пар, члены команды могут собираться в больших группах для неформального обсуждения идей проекта. Например, если в проекте разделены бэкенд- и фронтенд-разработки либо если API, предоставляющие и запрашивающие данные, разрабатываются разными командами, то целесообразно неформальное обсуждение проекта. Это может стать отличной возможностью добавить в обсуждение проблемы тестирования. Наблюдая за работой наших команд, мы иногда можем подключаться к этим неформальным обсуждениям.

4.3.3. Организация специальных встреч

Хотя использование регулярных церемоний или обсуждений в командах часто является самым быстрым путем при тестировании дизайна API, иногда эти варианты не позволяют достичь желаемых целей. Например, планирование спринта может быть слишком долгим, а работа в паре — не обеспечить желаемого качества обмена информацией. Тогда следует выбрать иной подход.

Принцип «Три амиго» предполагает встречу членов команды, чтобы обсудить идеи проекта и лучше разобраться в том, что им нужно сделать. Такая формальная или полуформальная встреча проводится, когда уже имеется необходимая информация для проектирования и разработки. Участники — тестировщик, разработчик и владелец продукта (к примеру). Цель встречи состоит в том, чтобы после ее окончания каждый ушел с четким единым пониманием того, что должно быть сделано.

Подход «Три амиго» прост в применении и может стать отличным способом стимулирования команды к участию в тестировании дизайна API. По мере развития этого подхода вы можете адаптировать его к своему контексту. Можно сделать такие встречи формальной частью цикла разработки программного обеспечения, добавив колонку «Анализ» на канбан-доску. Или можно оставить это мероприятие неформальным, чтобы члены команды обсуждали проблемы, только когда это необходимо.

Кроме того, можно увеличить число участников таких встреч до четырех, пяти или даже большего числа. Это позволяет взглянуть на ситуацию с разных сторон. Например, если разделены направления разработки, мы можем пригласить по одному представителю каждого направления. Также можно пригласить члена команды, ответственного за UX. Главное, что нужно помнить при работе с методом «Трех амиго», — это то, что он гибкий. Обсуждение проблем с коллегами всегда полезно.

4.4. ТЕСТИРОВАНИЕ ДИЗАЙНА API КАК ЭЛЕМЕНТ СТРАТЕГИИ

Тестирование дизайна может показаться необычным началом для изучения многочисленных техник тестирования API. Важно подчеркнуть динамический характер такого тестирования, благодаря чему оно впишется во множество различных ситуаций. Хорошая стратегия тестирования характеризуется целостностью. Она должна фокусироваться на разных аспектах — везде, где велики риски снижения качества продукта.

Если вспомнить модель, которую мы рассматривали в главе 1, то цель тестирования — убедиться в соответствии того, что мы хотим построить, тому, что мы строим. Но для этого необходимо знать, что именно мы хотим создать. Если мы неправильно понимаем, что хотим получить от команды, это может привести к несоответствию результата требованиям пользователей. Чтобы убедиться, что мы создаем то, что нужно, надо сосредоточить внимание на двух конкретных

областях, как показано на рис. 4.4: тестировать то, что мы знаем как команда (перекрытие двух областей), и то, что мы не знаем как команда, но что также имеет значение (остальная часть области «Воображение»).

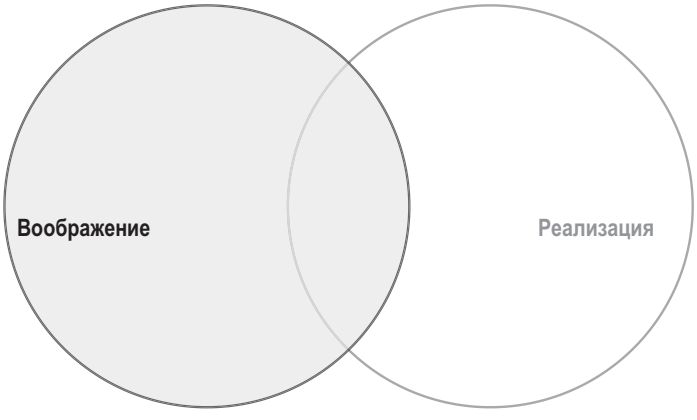

Рис. 4.4. Тестирование дизайна API для проверки соответствия планов и реализации

В отличие от других, этот вид тестирования фокусируется на проверке требований, проектов и идей. При правильном подходе вопросы, которые мы задаем, помогут команде выработать общее понимание того, что предлагается создать. Мы используем вопросы, чтобы прощупать проблемы, которые надо решить, разобраться, принесут ли наши решения пользу, и выявить проблемы и идеи, которые еще не обсуждались (например, отправка неверных кодов состояния). Чем больше мы будем задавать вопросов и обсуждать идеи и проекты, тем легче будет расширить область, которую мы знаем как команда, и сократить то, чего мы не знаем. В результате мы получим более информированную команду с четким представлением, как улучшить качество создаваемого продукта.

ИТОГИ

- Мы рискуем получить некачественный продукт, если не уделим время обсуждению и проверке требований и идей проекта.
- Заранее подвергая сомнению дизайн API, мы можем выявить проблемы на ранней стадии и развеять заблуждения.
- По своей сути тестирование — это постановка вопросов. Мы можем использовать эти навыки для проверки идей и проектов.

- Постановка вопросов требует хороших навыков критического и нестандартного мышления.
- Для определения нужных вопросов мы можем использовать такие приемы, как техника шести вопросов.
- Для расширения списка вопросов можно использовать такие методы, как вопросы с «еще», метод воронки, анализ типов данных и визуализация.
- Чтобы облегчить совместные обсуждения и тестирование, можно использовать современные инструменты документирования, такие как Swagger.
- Swagger использует схему OpenAPI 3, чтобы помочь точно и четко описать дизайн API.
- Дополнительные инструменты Swagger полезны для обмена документацией и быстрого создания кода.
- Участие в регулярных встречах и инициация специальных обсуждений дают прекрасные возможности для тестирования проектов.

5 Исследовательское тестирование API

В этой главе

- ✓ Что такое исследовательское тестирование и как оно работает
- ✓ Как планировать сеансы исследовательского тестирования
- ✓ Что включает в себя сеанс исследовательского тестирования
- ✓ Как поделиться результатами сеанса исследовательского тестирования

Прежде чем мы погрузимся в исследовательское тестирование, давайте начнем с небольшого задания. Найдите лист бумаги и ручку и напишите инструкцию, которая проведет читателя из точки А в точку Б. Это может быть путь от входной двери офиса до вашего рабочего места или от входной двери дома до кухни. Постарайтесь не слишком беспокоиться о маршруте. Когда вы напишете инструкцию, дайте ее кому-нибудь и попросите его начать в точке А и, используя ваши записи, добраться до точки Б. Когда он завершит путешествие, задайте простой вопрос: что он видел, следуя вашим инструкциям? Например, было ли что-то интересное на стенах? Был ли на каком-либо столе необычный предмет? Запишите ответ, а затем попросите того же человека повторить тот же путь, но

на этот раз без инструкций. Скажите, что у него есть 10 минут, чтобы добраться из пункта А в пункт Б, и в течение этого времени он может записывать все, что видит вокруг себя по мере продвижения.

Затем сравните наблюдения при использовании инструкций и без них. Вероятно, объем и детализация сведений во втором случае будет гораздо выше. Дело в том, что на втором этапе ваш доброволец проводил неформальный сеанс исследовательского тестирования. Во время этого сеанса он исследовал ограниченную область с целью узнать, что находится вокруг. Он продемонстрировал, как по своей сути работает исследовательское тестирование и в чем его преимущества. В этой главе мы рассмотрим сеанс исследовательского тестирования, узнаем больше об этом методе и о том, как исследовать целенаправленно и структурированно.

5.1. ЗНАЧЕНИЕ ИССЛЕДОВАТЕЛЬСКОГО ТЕСТИРОВАНИЯ

Многие слышали об исследовательском тестировании, но не все знают, как оно работает и в чем заключается. Некоторые ошибочно считают его ситуативным или бесструктурным подходом к тестированию, который трудно применим на практике или бесполезен. Но как мы узнаем, исследовательское тестирование — это достаточно системный подход, который подразумевает баланс между структурой и свободой, чтобы помочь «исследователю» получить как можно больше полезной информации.

5.1.1. Цикл тестирования в исследовательском тестировании

Глубокое погружение в исследовательское тестирование предлагает книга Элизабет Хендриксон (Elisabeth Hendrickson) *«Explore It!»*. В ней утверждается, что исследовательское тестирование обеспечивает следующие возможности:

> *...одновременно изучать систему, разрабатывая и выполняя тесты, и использовать обратную связь от последнего теста для проведения следующего.*

Имеется в виду, что во время исследовательского тестирования мы повторяем цикл много раз, как показано на рис. 5.1.

Рис. 5.1. Жизненный цикл теста, который начинается с фазы проектирования, а затем движется по часовой стрелке, чтобы начать новый цикл

Каждый шаг цикла работает следующим образом:

1. *Проектирование* — основываясь на имеющихся знаниях о тестируемой системе, определяем новые вопросы, которые хотели бы задать, чтобы заполнить пробелы в наших знаниях. Придумываем новую идею тестирования и разрабатываем ее.
2. *Выполнение* — выполняем разработанный тест. Допустимо использование дополнительных инструментов.
3. *Анализ* — анализируем полученные результаты.
4. *Изучение* — получив новую информацию, начинаем цикл снова.

Все это происходит в относительно быстром темпе, как правило, в нашей голове. Чтобы рассмотреть конкретный пример, давайте представим, что мы изучаем поле формы, которое было разработано для приема «правильных» телефонных номеров:

1. *Проектирование* — мы знаем, что поле формы должно принимать только действительные телефонные номера, поэтому придумали тест, который проверяет два правильных формата телефонных номеров из двух разных стран — Великобритании и США.
2. *Выполнение* — отправляем по одному корректному телефонному номеру для каждой страны.
3. *Анализ* — оказывается, что номер Великобритании принимается, а номер США — нет.
4. *Изучение* — узнав, что не все телефонные номера считаются «корректными», мы можем попробовать ввести номера других стран.

Такой цикл будет повторяться много раз за один сеанс исследовательского тестирования (подробнее о сеансах мы поговорим позже). Такой порядок позволяет

нам следовать шаблону, который обеспечивает необходимую структуру, но также дает свободу выбора задач и тестов. Однако это обоюдоострый меч. Как было указано ранее, мы не можем тестировать все подряд, поскольку время ограниченно. Мы должны убедиться, что наше тестирование приносит пользу, поэтому нам нужны рекомендации, на чем сосредоточиться, а что оставить на другой день.

5.2. ПЛАНИРОВАНИЕ ИССЛЕДОВАНИЯ

Мы узнали, что исследовательское тестирование является структурированным подходом, но откуда берется структура? Она приходит в форме уставов, определяющих конкретные, измеримые цели, которых мы надеемся достичь в ходе проведения сеансов исследовательского тестирования. Например, устав может быть следующим:

- изучите `branding` API...
- ...с различными деталями B&B...
- ...чтобы узнать, правильно ли `branding` API сохраняет данные B&B.

В качестве альтернативы можно использовать следующий формат, которым поделился мой коллега Дэн Эшби (Dan Ashby):

- посмотрите на обновление в `branding` API...
- ...для выявления проблем при сохранении деталей брендинга.

Цель устава — четко сформулировать, чего мы надеемся достичь в ходе исследовательского тестирования, не слишком формализуя его. Например, для приведенного выше устава мы можем придумать тесты для названий, местоположений и адресов отелей B&B, чтобы убедиться, что эти идеи соответствуют цели устава. Мы также знаем, что если проводим тесты, которые рассматривают коды ответов `booking` API, значит, отклонились от цели и, возможно, не обнаружим информацию, полезную для нашей команды.

5.2.1. Создание уставов

Когда мы думаем об уставах, в воображении возникают образы знаменитых исследователей, отправляющихся на поиски приключений в неизвестные земли или древние руины. Потому что уставы — это заявления, описывающие то, что исследователь стремится обнаружить, и то же самое можно сказать об

исследовательском тестировании. Подобно тому как исследователь создает устав для открытия неизвестной области мира, мы используем уставы для открытия рисков. Мы определяем риски, а затем фиксируем их в уставах, чтобы использовать в сеансах исследовательского тестирования.

Мы уже подробно рассматривали анализ рисков в главе 3. Но вкратце напомним, что мы можем определить риски, вооружившись скептическим мышлением, с помощью таких инструментов, как RiskStorming. При этом нужно подвергать сомнению наши предположения и моделировать ситуации, которые могут негативно повлиять на качество продукта. В ходе этого процесса мы выявим риски, которые могут повлиять на наш продукт, и именно эти риски можно превратить в уставы, которые будут использоваться в исследовательском тестировании.

Например, давайте рассмотрим риск, который мы определили в главе 3, сформулировав его применительно к платформе API restful-booker:

- *Администратор не может использовать* `branding` *API.*

Это может быть перефразировано следующим образом:

- *Конечной точке* `PUT /branding` *не удается обновить данные в* `branding` *API.*

У нас есть краткое, конкретное описание риска, который мы хотим проверить, но в текущей форме он не поддается измерению и не совсем конкретен. Именно здесь может помочь устав. Например, давайте обновим наш риск, используя общий шаблон устава, который был популяризирован в *«Explore It!»*:

```
Изучение <Цель>
Использование <Инструменты>
Обнаружение <Риски/Дополнительная информация>
```

Используя этот подход, мы можем перевести наш риск в следующий устав:

```
Изучение конечной точки API PUT /branding/
Использование различных наборов данных
Обнаружение проблем, при которых данные обрабатываются некорректно
```

Преобразуя риск в устав, мы получаем более четкую цель, которую надеемся достичь в сеансе исследовательского тестирования. Теперь мы знаем, на чем сосредоточиться и какие инструменты можем использовать. Устав помогает начать тестирование, а также определить успешность его результатов.

Для создания уставов мы определяем риски, а затем преобразуем их формулировки с помощью шаблона, который описывает наши намерения во время сеанса исследовательского тестирования. Как и другие техники тестирования,

составление уставов требует практики. Помогают в создании уставов шаблоны, такие как шаблон Дэна Эшби, который мы видели ранее:

```
Посмотреть на <Цель>
Для тестирования <Идеи тестирования>
```

Мы можем записать это следующим образом:

```
Посмотрите на PUT /branding/
Для тестирования проблем, при которых данные сохраняются неправильно
```

Когда я только начинал работать с уставами, то считал особенно полезным шаблон Майкла Д. Келли (Michael D. Kelly), которым он поделился на вебинаре *«Tips for Writing Better Charters for Exploratory Testing Sessions»* (https://youtu.be/dOQuzQNvaCU):

```
Моя задача — проверить <Риск> на <Охват>.
```

Мы можем записать это следующим образом:

```
Моя задача — проверить, не сохраняются ли данные для конечной точки PUT /
branding/API
```

Какой формат использовать — решать вам. В конечном счете цель состоит в том, чтобы определить серии уставов, отражающих различные риски, которые мы хотим изучить, чтобы планировать, какие исследовательские тестирования проводить и когда.

Упражнение

Выберите один из рисков, определенных во время работы с RiskStorming в главе 3, либо придумайте другой риск для исследования в песочнице API. Составьте один или несколько уставов, используя шаблоны, которые мы изучили, и определите основные направления сеанса исследовательского тестирования.

Моменты выбора уставов

Уставы могут быть созданы как во время формальных встреч, таких как планирование спринта или обсуждение пользовательских историй, так и при проведении сеансов исследовательского тестирования. Наиболее эффективно сочетание обоих подходов. Время, когда команда находится вместе, удобно для создания различных уставов. Но не бойтесь добавлять их и когда проводите исследовательское тестирование или делитесь своими открытиями с другими.

5.2.2. Уставы и сеансы исследовательского тестирования

После составления устава следующим шагом будет планирование сеансов исследовательского тестирования. Сеанс исследовательского тестирования — это однократное проведение исследовательского тестирования в течение определенного времени в соответствии с уставом. Например, мы можем запланировать проведение одного сеанса для определенного нами устава, как показано на рис. 5.2.

Рис. 5.2. Схема при отношении «один к одному» между уставами и сеансами исследовательского тестирования (ИТ)

Тем не менее иногда может быть более выгодно провести более одного сеанса исследовательского тестирования, как показано на рис. 5.3.

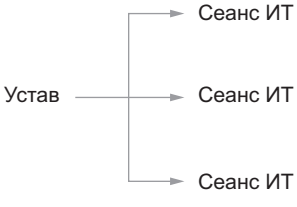

Рис. 5.3. Схема при отношении «один ко многим» между уставами и сеансами исследовательского тестирования

В этом случае каждый сеанс исследовательского тестирования будет отличаться от другого. Знания о том, что мы тестируем, навыки тестировщика и многие другие факторы будут влиять на ход процесса и результаты. Если провести несколько сеансов, посвященных одной и той же задаче, то в итоге мы, скорее всего, получим интересные открытия, которыми сможем поделиться с коллегами.

5.2.3. Проведение исследовательского тестирования

Число сеансов исследовательского тестирования для одного устава, а также приоритеты наборов уставов зависят от времени, которым мы располагаем. Уставы не просто направляют наше тестирование, но и являются способом организации исследования и его порядка. Например, допустим, что для платформы restful-booker мы определили следующие три устава в порядке убывания приоритетов:

1. Исследуйте `Create Room` API , чтобы обнаружить риски, связанные с неправильным созданием комнат (номеров).
2. Исследуйте `Delete Room` API, чтобы обнаружить риски, связанные с неправильным удалением комнат (номеров).
3. Исследуйте `Create Room` API , чтобы обнаружить риски, связанные с отправкой некорректных ответов.

В идеале мы хотели бы провести как минимум три сеанса исследовательского тестирования для второго устава, но у нас есть время только на четыре сеанса для всего списка. Варианты могут быть следующими:

- Проведите один сеанс для первого устава и три для третьего. Тогда результаты для третьего устава останутся неизвестными, и можно будет поговорить о выделении дополнительного времени или принятии риска неизвестности.
- Проведите по одному сеансу для каждого устава в порядке приоритета, после чего вернитесь ко второму уставу и проведите дополнительный сеанс исследовательского тестирования.

Оба варианта имеют свои преимущества и недостатки. Однако смысл этого примера в том, чтобы показать, как мы можем использовать уставы для организации исследовательского тестирования. Создавая уставы, мы определяем не только способы управления тестированием, но и способы организации работы, а также обмена информацией о том, что может быть сделано, что уже сделано или будет сделано, что может быть использовано для информирования о ходе исследовательского тестирования.

Уставы и регрессионное тестирование

Уставы можно использовать для организации сеансов исследовательского тестирования при регрессионных тестах. Если мы сохраняем уставы таким образом, что можем отслеживать, какие из них выполнялись и когда, а также на чем они фокусировались, то это поможет принять решение о том, какое регрессионное тестирование следует провести. Например, мы можем решить заново запустить уставы, которые не запускались в течение определенного промежутка времени. Либо можем выбрать уставы, относящиеся к определенным областям системы, которые нас беспокоят, потому что на них могли повлиять некие недавние изменения.

5.3. ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ: ПРИМЕР

Поскольку исследовательское тестирование позволяет человеку проводить сеанс так, как он хочет (в рамках разумных ограничений), это усложняет обучение этой технике. Как мы уже говорили ранее, каждый сеанс исследовательского тестирования будет отличаться от другого. Различия могут быть как тонкими, так и кардинальными, но какими бы они ни были, мы не можем просто скопировать то, что делал кто-то другой.

Однако существуют шаблоны и техники, которые могут быть использованы независимо от того, что мы исследуем. Итак, чтобы помочь вам изучить различные аспекты исследовательского тестирования, проанализируем уже проведенный сеанс и определим, что происходило в его ходе, какие техники и инструменты применялись, почему мы их выбрали и какая мысль стоит за каждым из них.

Для нашего примера рассмотрим сеанс исследовательского тестирования, который я проводил на основе следующего устава:

- Изучение возможности бронирования номера.
- Использование различных наборов данных.
- Обнаружение проблем, которые могут привести к сбою или ошибке при бронировании.

Мы начнем с разбора идей, которые возникли в процессе исследовательского тестирования. В этой части я буду ссылаться на определенные разделы моих заметок о тестировании. Если вам нужно больше контекстной информации, просмотрите полные записи тестирования в виде диаграммы связей на GitHub (http://mng.bz/1oDV).

5.3.1. Начало сеанса

Один из самых сложных аспектов исследовательского тестирования — определить, с чего начать. Обычно мы находимся в одном из двух состояний: либо знаем очень мало о том, что собираемся тестировать, либо знаем слишком много и сомневаемся, с чего начать. Я считаю, что лучше сначала потратить некоторое время на изучение того, что мы собираемся тестировать, а уже потом углубляться в идеи тестирования.

Например, в сеансе исследовательского тестирования запроса на бронирование номера я начал с создания заметок в форме диаграммы связей и добавил

некоторую информацию о сеансе, например, когда я начал, какую версию тестирую и в каком окружении. Также я добавил детали HTTP-запроса, который используется для успешного бронирования, включая URL, HTTP-заголовки и HTTP-тела (рис. 5.4).

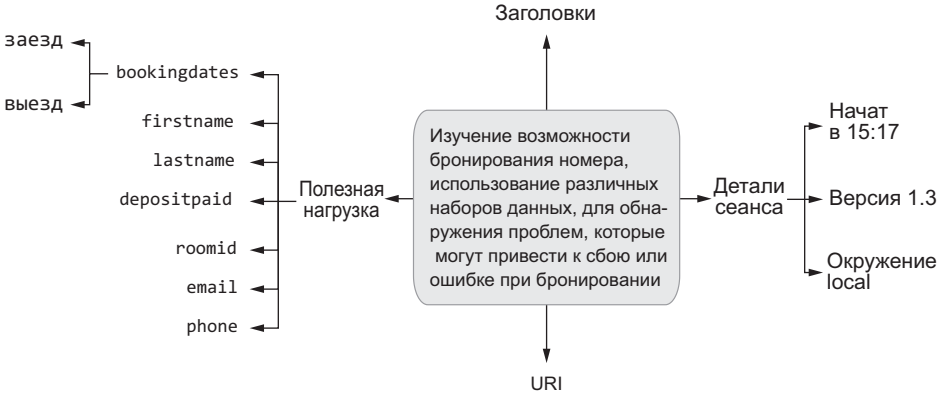

Рис. 5.4. Записи, сделанные в начале сеанса исследовательского тестирования

Я основывался на данных, которые узнал с помощью техник, описанных в главе 2, в частности, используя инструменты разработчика для захвата `POST /booking HTTP`, который я затем скопировал в инструмент тестирования Postman. Однако я мог узнать об этом запросе и из документации, анализа исходного кода или беседы с человеком, который создал конечную точку веб-интерфейса API.

Получив эту информацию, я мог приступить к тестированию. На рис. 5.5 показана серия идей для тестирования объекта `bookingdates` в полезной нагрузке HTTP body, которые сформулированы в форме вопросов.

У меня было понимание того, для чего используются даты бронирования при его создании. Изучив и осмыслив эту информацию, используя критическое и нестандартное мышление, я определил тестовые идеи. Как мы помним из главы 4, критическое мышление позволяет глубже изучать идеи и концепции, а нестандартное мышление позволяет их расширять. Я использовал эти навыки, чтобы задавать такие вопросы, как, например:

- Что делать, если даты заезда или выезда приходятся на високосный год?
- Что, если формат даты был другим?
- Что, если я добавлю еще какие-нибудь неправильные данные?

Рис. 5.5. Заметки о тестировании, связанном с датами заезда и выезда из отеля

Каждый из этих вопросов дает возможность узнать новые подробности о приложении, разработав тест и наблюдая за ходом и результатами его выполнения. Пример:

- Установка даты заезда или отъезда на 29-02-2020.
- Ввод дат в форматах 01/01/2000 или 2000-30-12.
- Изменение дат на null, целочисленные или строковые значения.

Исследовательское тестирование с помощью GraphQL

Хотя структура GraphQL отличается от REST, с которым мы работаем в нашем примере, подход к исследовательскому тестированию схож. Возьмем простой запрос `Character`:

```
query Character {
  character(id: 1) {
    name
    created
  }
}
```

Мы можем применить уже знакомый подход, рассмотрев, какие еще поля мы можем запросить и какие идентификаторы и фильтры можем добавить. Например, для `Character id`, который принимает целое число, я могу задать следующие вопросы: что, если я запрошу несуществующие символы? Что, если я удалю значение? Что, если я предоставлю другой тип данных?

Упражнение

Придумайте несколько идей/вопросов для тестирования дат заезда и выезда. Запишите каждую из идей, мы используем их позже.

5.3.2. Как понять, когда что-то не так

Новички, приступающие к исследовательскому тестированию, часто сомневаются, какие проблемы следует рассматривать, а какие нет. В отличие от сценарных подходов, разработанных специально для тестирования в соответствии с требованиями или критериями приемки, исследовательское тестирование не имеет четких паттернов. Человек, проводящий исследовательское тестирование, сам решает, стоит ли что-то рассматривать как проблему, и это может вызвать затруднения у начинающих.

Чтобы начать обнаруживать проблемы, которые нарушают ожидания, используются оракулы. Оракул — это источник информации, которая может быть явной, например требования, или неявной, например правила языка. Оракулы помогают определить, является ли новая информация нормальной или свидетельствует о потенциальных проблемах, требующих решения. Давайте рассмотрим несколько примеров оракулов, которые я использовал в процессе исследовательского тестирования для обнаружения проблем.

- *Продуктовый оракул* — мы используем наши знания о системе, чтобы определить ее предсказуемость. Например, во время тестирования я обнаружил, что иногда ошибка 400 возвращалась с дополнительным комментарием (например, что поле в полезной нагрузке не может быть пустым), а в некоторых случаях — без какого-либо объяснения. По моему мнению, приложение всегда должно возвращать код 400 с четким сообщением об ошибке. Непоследовательность логики продукта нарушает мой продуктовый оракул, что указывает на потенциальную проблему. Какое сообщение об ошибке должно быть отправлено, общее или содержащее комментарий? Это требует дальнейшего исследования.

- *Нормативный оракул* — многие проекты подчиняются правилам, нормам и законам, которые необходимо соблюдать. В качестве примера можно привести медицинское программное обеспечение, соответствующее определенным нормам, финансовые продукты, соответствующие финансовым правилам, или, что однажды встретилось в моей практике, архитектурные стандарты. Когда я тестировал указание дат заезда и выезда, обнаружилась проблема: при установке дат регистрации и выписки в один и тот же день возникала ошибка 409. Рассмотрим определение кода состояния 409 из rfc7231:

  Код состояния 409 (конфликт) указывает на то, что запрос не может быть выполнен из-за конфликта с текущим состоянием целевого ресурса.

  Для нашей проблемы это не подходит. Конфликта нет, потому что нет бронирования, с которым можно было бы конфликтовать, поэтому 409 — в данном случае неправильный код статуса.

- *Оракул осведомленности* основан на нашем опыте и знаниях о системах, выходящих за рамки тестируемого продукта. Наша способность выявлять проблемы основывается не только на знаниях о тестируемом продукте, но и на накопленном опыте использования ПО, с которым мы работаем профессионально или используем в свободное время. Именно эти знания помогли мне установить, что при попытке бронирования возвращается ошибка 500, если `message` API недоступен. Проблема здесь не в ошибке 500, а в том, что бронирование фактически сохраняется. Мой оракул осведомленности интерпретировал ошибку 500 как что-то, что не удалось завершить. Однако в данной ситуации процесс бронирования был частично завершен, что не соответствовало моему прошлому опыту работы с другими API, которые выдавали ошибку 500.

Эти оракулы демонстрируют, что в процессе наблюдения за нашими системами есть различные способы интерпретации информации для определения проблем (помимо более распространенного подхода сравнения того, что мы видим, с письменными требованиями). Для того чтобы комфортно работать с оракулами, требуется время и практика, прежде чем это станет привычкой. Поэтому для начала полезно иметь перед собой список оракулов в качестве краткого справочника, который поможет вызвать идеи в виде эвристики — техники, которую мы рассмотрим далее.

Совместная работа над ошибками

Более подробно о том, что мы узнали в ходе тестирования, мы поговорим позже, но как минимум я всегда советую вам хотя бы поговорить со своей командой об обнаруженных ошибках. Оракулы ошибаются, то есть они работают только

в определенных ситуациях, и иногда это означает, что мы поднимаем проблемы, которые на самом деле таковыми не являются. Вот почему полезно поговорить с командой после сеанса тестирования, чтобы подтвердить, какие из них действительно являются проблемами. Разговор с разработчиком или владельцем продукта поможет определить, что нужно исправить, а также продемонстрировать другим значимость нашей работы.

Упражнение

Существует еще больше оракулов, которые вы можете использовать. Их хорошо обобщил Майкл Болтон (Michael Bolton) в своей статье «FEW HICCUPPS» (http://mng.bz/5QrB/). В качестве упражнения прочитайте каждый из оракулов, а затем придумайте пример проблемы или ошибки, которая описывает, как работает оракул.

5.3.3. Придумывание идей для тестирования

Отсутствие идей может быть сигналом к тому, что сеанс тестирования пора завершать, и мы рассмотрим эту ситуацию более подробно ниже. Однако благодаря использованию эвристики можно продолжать расширять область тестирования, рождая новые идеи.

Ричард Брэдшоу (Richard Bradshaw) и Сара Дири (Sarah Deery) в своей статье *«Software Testing Heuristics: Mind The Gap!»* (http://mng.bz/6XVo) высказали следующую мысль:

> Эвристики — это когнитивные короткие пути. Мы используем их в условиях неопределенности, зачастую автоматически и бессознательно... для быстрого решения проблем и принятия решений.

Мы каждый день используем эвристику для решения проблем как сознательно, так и бессознательно. Например, у меня есть личная эвристика, которую я применил сознательно, чтобы попасть в свой сарай, закрытый на замок. У меня есть два ключа на красном брелоке — один от замка сарая, а второй — от другой двери. Проблема в том, что они выглядят почти одинаково, поэтому я постоянно пытался вставить в замок не тот ключ. Однако я разработал эвристику на основе того факта, что один ключ слегка изогнут, а другой — нет. Эвристика — это фраза «Красное — это правильно».

Всякий раз, когда ключи оказывались у меня в руках, фраза «Красное — это правильно» заставляла меня вспомнить, что ключ от сарая не деформирован.

Произнося ее, я понимал, какой ключ выбрать. Эта фраза интересна по нескольким причинам. Во-первых, она демонстрирует ошибочную природу эвристики, потому что фраза работает только в случае, если у меня есть красный брелок. Если бы у меня был запасной зеленый брелок, это не сработало бы. Таким образом, эвристика помогает мне решить только конкретную проблему, так же как и другие эвристики предназначены для решения других конкретных проблем. Вторая причина заключается в том, что в какой-то момент я перестал повторять эту фразу, поскольку достиг точки, когда уже твердо знал, что неповрежденный ключ на красной связке — правильный. Эвристика стала бессознательной, потому что я практиковал ее много раз. Я не сомневаюсь, что все еще использую эту эвристику, но где-то глубоко в подсознании.

Эти же соображения применимы к использованию эвристики для придумывания идей тестирования. Многие из моих идей тестирования дат, их форматов и типов данных возникли благодаря внутренней эвристике, которую я выработал, тестируя поля дат в прошлом. Чем больше я тестировал, тем больше внутренних эвристик у меня появлялось. Однажды, как уже говорилось ранее, наступает момент, когда мы исчерпываем все эти идеи и эвристики. Но вместо того чтобы останавливаться, мы можем переключиться и начать использовать явно выраженные эвристики, чтобы помочь выявить новые идеи.

Например, отличным ресурсом, который я часто использую в сеансах исследовательского тестирования, является шпаргалка по эвристике тестирования (http://mng.bz/Ay5K), созданная Элизабет Хендриксон (Elisabeth Hendrickson), Джеймсом Линдсеем (James Lyndsay) и Дейлом Эмери (Dale Emery). В шпаргалке собран список эвристик и атак на типы данных, которые помогают появлению идей тестирования. Я использовал их для получения следующих новых идей:

- имитация пустых имен и фамилий путем заполнения полей пробелами;
- использование символов с ударениями и эмодзи;
- использование неправильных дат, например 30 февраля.

Подсказки из шпаргалки рождают идеи, которые открывают новые пути для изучения, в свою очередь создающие новые идеи.

Мнемоники также часто используются в качестве эвристики для появления идей. Я использовал следующие аббревиатуры, чтобы генерировать больше идей для тестов:

- *BINMEN*, созданная Гвен Диаграм (Gwen Diagram) и Эшем Винтером (Ash Winter). Она составлена из первых букв слов *границы* (boundaries), *недействительные записи* (invalid entries), *нули* (nulls), *метод* (method), *пусто* (empty), *отрицательные числа* (negatives), каждое из которых может быть использовано для запуска различных идей. Когда я просматривал аббревиатуру BINMEN, в глаза бросилось слово «метод», потому что это я еще не исследовал. В результате появились идеи, связанные с изучением того, какие методы HTTP доступны для конечной точки `/booking/`.
- *POISED*, которая расшифровывается как *параметры* (parameters), *вывод* (output), *совместимость* (interoperability), *безопасность* (security), *исключения* (exceptions) и *данные* (data). Ее автор — Амбер Рейс (Amber Race). Слово «исключения» вызвало идеи, связанные с обработкой ошибок. Соответствуют ли возвращаемые ошибки тому, что я ожидаю увидеть (вспомните наш оракул осведомленности)? Полезны ли сообщения об ошибках? Легко ли устранить ошибки?

Для генерации новых идей можно использовать гораздо больше эвристик, и они очень удобно собраны в полезные коллекции, например, такие:

- коллекция Линн Макки (Lynn McKee) на сайте *Quality Perspectives* (http://mng.bz/ZANO);
- эвристическая диаграмма связей Дела Дьюара (Del Dewar) (http://mng.bz/R4z0);
- периодическая таблица тестирования Эди Стоукса (Ady Stokes) (https://www.thebigtesttheory.com).

Важно помнить, что инструменты эвристики — это не упражнения по проверке списков. Не существует единственно правильного способа использования эвристик; важно наше отношение к ним. Если они вызывают идеи, используйте их. Если нет, оставьте их и двигайтесь дальше.

Упражнение

Вернитесь к идеям тестов, которые вы предложили для раздела о датах запроса на бронирование. Просмотрите ресурсы, о которых мы узнали, например шпаргалку по эвристике тестирования, и используйте их для создания новых идей.

5.3.4. Использование инструментов

Мы изучили техники, позволяющие придумывать новые идеи в ходе исследовательского тестирования. Теперь давайте посмотрим, какие программные инструменты помогают тестировать больше, глубже и быстрее. Я использую целый набор таких инструментов (рис. 5.6).

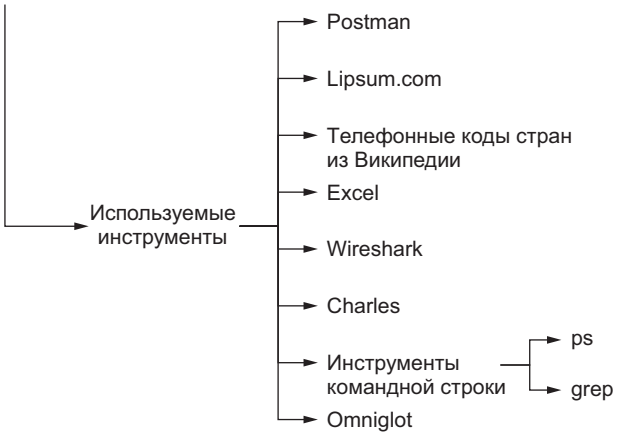

Рис. 5.6. Задокументированный список инструментов, использованных в сеансе исследовательского тестирования

Как мы видим, существует ряд инструментов, которые служат разным целям. Давайте рассмотрим, как они использовались и как помогли улучшить сеанс исследовательского тестирования.

Excel и HTTP-клиенты

Первый пример связан с полем телефона в теле HTTP-запроса. Во время сеанса мне пришла в голову идея попробовать различные форматы телефонных номеров, чтобы проверить, будет ли валидирован каждый из них. С этой идеей возникли следующие проблемы:

- Мне нужен был список международных номеров.
- Существует множество вариантов международных номеров, которые можно опробовать.

Первая проблема была решена с помощью Excel и Википедии. Форматы международных номеров представлены в таблице на странице Википедии. Я взял эти

данные, добавил их в таблицу Excel и с помощью формул создал случайные номера, соответствующие каждому формату, подобные приведенным в табл. 5.1.

Таблица 5.1. Пример таблицы с данными

ID	Страна	Код	Длина	Пример номера
1	Афганистан	93	9	93212264949
2	Аландские острова	358	10	3583415884081
3	Албания	355	9	355224314764
4	Алжир	213	9	213941947247

Затем надо было попробовать каждый из них в тесте. Работа «вручную» была бы слишком медленной, поэтому я воспользовался инструментом Postman.

1. Я перехватил HTTP-запрос `POST /booking/` и добавил его в коллекцию в Postman.
2. Затем я обновил тело JSON, чтобы превратить значения параметров `id` и `phone` в переменные Postman (обратите внимание, что имена переменных и строк в CSV-файле совпадают):

```
{
    "bookingdates": {
         "checkin": "2022-12-01",
         "checkout": "2022-12-04"
    },
    "depositpaid": true,
    "firstname": "Mark",
    "lastname": "Winteringham",
    "roomid": {{id}},
    "phone" : "{{example}}",
    "email" : "test@test.com"
}
```

С помощью инструмента runner в Postman я импортировал CSV-файл и запустил коллекцию с измененным запросом несколько раз. На каждой итерации из файла CSV извлекалась новая строка и перед отправкой запроса вставлялась в тело JSON. Подробнее об использовании этой функции можно узнать на сайте Postman (http://mng.bz/2nld).

При отправке каждого запроса вводился новый номер телефона, а ответ сохранялся. По результатам теста оказалось, что поддерживаются не все международные номера, что указывает на потенциальную проблему.

Упражнение

Аналогичную идею тестирования можно применить к параметру `email` в запросе `POST /booking/`. Используя инструмент runner в Postman и CSV-файлы, проведите эксперимент, чтобы протестировать различные форматы электронной почты, как допустимые, так и недопустимые. После окончания эксперимента проанализируйте результаты.

Прокси и mock

Во втором примере я использовал комбинацию инструментов прокси и mock. Как мы знаем из главы 2, прокси-инструмент Wireshark можно использовать не только для мониторинга HTTP-трафика между пользовательским интерфейсом и бэкендом API (подобно другим прокси-инструментам), но и для мониторинга трафика между разными API. В данном случае с помощью Wireshark была обнаружена связь между `booking` API и `message` API, как показано на рис. 5.7.

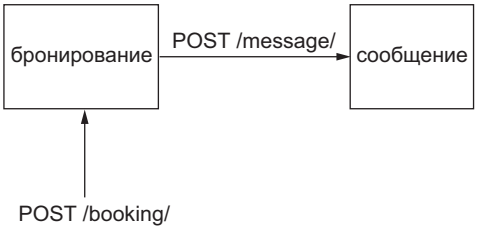

Рис. 5.7. Модель, описывающая взаимосвязь между `booking` API и `message` API

Использование знаний о системе в целом

В этом примере я указал, что использовал Wireshark для обнаружения взаимосвязи между `booking` и `message` API, но мы могли бы легко сделать это в рамках первоначального исследования продукта в главе 2. Хорошее знание системы могло бы помочь нам прийти к выводам, аналогичным тем, что были обнаружены в данном примере.

Зная о существовании такой взаимосвязи, я придумал новую идею теста — проверить, что произойдет, если `message` API отправит обратно неожиданную ошибку или неверный код статуса. Однако у меня возникла проблема с отправкой этих неожиданных ошибок и неправильных кодов состояния в API бронирования. Поэтому потребовался еще один инструмент — WireMock.

С помощью WireMock можно имитировать поведение запроса и ответа веб-интерфейса API, что дает больше контроля над информацией, передаваемой между API. В моем тесте я использовал WireMock для имитации `message` API и отправил серию различных кодов состояния обратно в `booking` API. Для этого я загрузил автономную версию WireMock, которую можно найти в документации (http://wiremock.org/docs/running-standalone/). После загрузки WireMock сначала нужно было завершить текущий запущенный `message` API — найти его PID-номер и выполнить команду `kill {pid_number}`. WireMock был запущен с помощью следующей команды:

```
java -jar wiremock-jre8-standalone-2.28.0.jar --port 3006
```

> **ПРИМЕЧАНИЕ** Номер версии мог измениться. Обязательно проверьте правильность номера.

Эта команда запустила WireMock и настроила его на работу с `message` API благодаря параметру `--port 3006`, указывающему номер порта `message` API. WireMock создал папку `mappings`, в которую я добавил следующий JSON-файл:

```
{
  "request": {
    "method": "GET",
    "url": "/message"
  },
  "response": {
    "status": 400
  }
}
```

Это настроило WireMock вернуть код 400 в ответ на запрос `GET /message`. Я перезапустил WireMock и с загруженной связкой отправил запрос `POST /booking/`, чтобы наблюдать за реакцией `message` API на код состояния 400. Задавая в JSON различные коды состояния и ошибки подключения, удалось обнаружить, что когда `booking` API отвечает ошибкой 500, бронирование, тем не менее, сохраняется.

Упражнение

Аналогичная связь существует между `room` API и `auth` API, когда вызов `POST /room` отправляет дополнительный вызов `POST /auth/validate` в `auth` API, чтобы определить, можно ли создать комнату. Попробуйте сымитировать `auth` API с помощью WireMock, чтобы отправлять в `room` API различные коды состояния.

Эти два примера показывают, что с помощью автоматизированных инструментов можно реализовывать сложные идеи тестирования, которые дают интересные результаты. При проведении исследовательского тестирования полезно использовать возможности современных инструментов для поддержки и расширения тестирования. Можно создать набор таких инструментов, которые в дальнейшем пригодятся не только для исследовательского тестирования.

5.3.5. Ведение записей

Мы изучили процесс выявления и применения различных идей тестирования. Следует помнить, что в определенный момент нужно будет поделиться своими находками с командой. Скоро мы обсудим варианты обмена полученными знаниями, но когда дело дойдет до обмена деталями, в любом случае понадобится некая форма заметок.

Ведение заметок при исследовательском тестировании необходимо не только чтобы передать узнанное, но и чтобы контролировать тестирование и генерировать новые идеи. Пока мы уверены, что сделанные записи помогают контролировать и делиться результатами тестирования, способ их ведения определяем мы сами. Ниже описаны несколько распространенных подходов, которые используются специалистами по исследовательскому тестированию.

Майндмэп

Для начала обсудим формат, который я выбрал для рассмотренного сеанса. Построение диаграммы связей (майндмэп, интеллект-карты) начинается с создания корневого узла, к которому я добавил устав сеанса, а затем разветвления по различным направлениям исследования. Ценность этого подхода заключается в том, что по мере обнаружения новой информации ее можно записывать в узлы, которые подчеркивают связь между проведенным тестированием и обнаруженной информацией. Во время сеанса я добавлял на диаграмму полученные данные, затем вопросы и идеи тестирования, которые, в свою очередь, превращались в дополнительную информацию. По мере тестирования диаграмма связей органично развивалась, помогая отслеживать найденные сведения и решать, над чем работать дальше.

Если вы посмотрите на записи тестирования (http://mng.bz/mOor), то увидите, что диаграмма связей была составлена с помощью специальной программы, которая позволила сделать следующее:

- отметить различные направления диаграммы разными цветами;
- выделить узлы с ответами на вопросы одним цветом (зеленым), а узлы с вопросами, требующими дальнейшего рассмотрения, — другим (желтым);
- добавить значки, чтобы выделить обнаруженные факты, например ошибки.

Польза этих инструментов форматирования проявляется, когда делается снапшот результатов тестирования. Например, при беглом взгляде на записи можно увидеть множество красных узлов, означающих, что в ходе сеанса было обнаружено большое количество потенциальных проблем.

Критики диаграмм связей указывают, что они представляют собой просто перечень (чек-лист) в другом формате. Поэтому диаграмма связей, в которой не хватает деталей, выглядит как список проведенных тестов. Чтобы диаграммы связей отражали как ход тестирования, так и ваши мысли, используйте инструменты форматирования (разные цвета для разных событий) и фиксируйте дополнительную информацию. Попробуйте ограничить количество слов для каждого узла. Мне однажды посоветовали интересный прием: использовать не более трех слов в узле, чтобы стимулировать развитие мышления.

Ручка и бумага

Порой отвлечение от программ или экрана может принести пользу, поэтому многие люди предпочитают пользоваться ручкой и бумагой. Лично я считаю, что переключение внимания с экрана на бумажные заметки очень полезно для переключения мышления с наблюдения за системой на формулирование идей. И есть что-то приятное в том, чтобы писать заметки хорошей ручкой.

Кроме того, при использовании ручки и бумаги есть большая свобода выбора способа ведения заметок. Мы можем взять формальный подход, например метод Корнелла, который делит страницу на части: основные заметки, ключевые слова и комментарии, резюме. Другие предпочтут более наглядные средства. Моделирование, которое мы рассматривали в главе 2, является одной из форм ведения записей, помогающей организовать мысли и зафиксировать идеи в наглядном виде, чтобы ими было легко поделиться. Некоторые идут дальше, применяя технику «скетчноутинг», когда для документирования тестирования используются картинки и значки вместе с короткими фразами. Этот подход удобен, чтобы делиться не только результатами тестирования, но также мыслями и чувствами.

Каждый подход требует навыков ведения записей и дисциплины. Надо убедиться, что выдерживается правильный баланс деталей. Слишком много деталей — и мы теряем фокус, что приводит к сбоям. Слишком расплывчато — и потом будет сложнее поделиться узнанным. Кроме того, детальное описание сеанса тестирования с помощью скетчноутинга требует практики, а диаграммы должны быть выполнены таким образом, чтобы их было легко интерпретировать впоследствии.

Запись экрана

Последний подход — захват экрана в формате видео, которое затем можно просматривать. Некоторые считают это полезным, так как можно точно зафиксировать не только наблюдаемое, но и всю информацию, полученную при тестировании. При этом решается проблема «функциональной закрепленности» — когнитивного искажения, при котором мы настолько сосредоточены на наблюдении одного события, что полностью упускаем другое.

Просмотр записи позволяет обнаружить новые детали и проанализировать ход тестирования. Однако это трудоемкий процесс, особенно если сессия длится несколько часов. Кроме того, запись содержит только то, что отображается на экране, и не фиксирует мыслительные процессы, лежащие в основе сеанса, что затрудняет анализ качества тестирования.

Экспериментируйте с ведением записей

Правильный подход к ведению заметок может значительно улучшить результаты исследовательского тестирования. Идеален метод, который не отвлекает нас во время тестирования, но в то же время позволяет зафиксировать нужное количество информации, чтобы позже ею можно было поделиться. Экспериментируйте, чтобы определить наилучший подход для себя, пробуйте разные стили ведения записей и анализируйте, что работает, а что нет. В конце концов вы найдете оптимальный подход.

5.3.6. Уметь остановиться

В какой-то момент сеанс исследовательского тестирования должен закончиться. Но как определить точку, в которой нужно остановиться? Наша цель в ходе сеанса — узнать как можно больше информации, связанной с поставленной задачей. Однако мы сами должны определить момент, когда можно считать, что вся необходимая информация собрана. При этом можно использовать несколько техник или сигналов.

Закончились идеи для тестирования

Ранее мы обсуждали, как выявлять идеи тестов с помощью подсознательной эвристики, а затем использовать эвристику явно для генерации других идей. Но наступит момент, когда эвристики перестанут приносить плоды. Типичный признак — мы обнаруживаем, что наш ум уже не держит концентрацию на проблеме или повторяет старые идеи тестов (возможно, с незначительными изменениями). Обычно это означает, что мы исчерпали свои идеи и больше ничего не придумаем. Это совершенно нормально. Помните, что мы никогда не сможем проверить всё. Как мы обсудим позже, после сеанса надо поразмышлять над тестированием, и такой анализ может выявить новые идеи. Если это произошло, мы просто проводим еще один сеанс.

Истощение

Исследовательское тестирование требует больших умственных усилий, и в какой-то момент мы начнем снижать темп работы. Усталость и недостаток энергии обычно являются признаком того, что пора остановиться. Даже если мы чувствуем, что еще есть чем заняться, существует риск, что, чувствуя усталость, мы упустим что-то в наблюдениях. Вполне нормально остановиться, поразмыслить, а при необходимости провести новые сеансы позже. И всегда можно использовать записи с предыдущих сеансов в качестве подспорья для будущих.

Тайм-боксинг

Тайм-боксинг[1] полезен по нескольким причинам. Прежде всего, это помогает предотвращать усталость благодаря разделению сеансов на отрезки с регулярными перерывами между ними. В таком режиме удобно управлять приоритетами и расставлять акценты в случае нехватки времени.

Тайм-боксинг сеансов — это подход, который был популяризирован техникой управления сеансами исследовательского тестирования, которую создали Джеймс и Джон Бахи (James and Jon Bach) (http://mng.bz/PnY9). Сеансам могут быть выделены малые, средние и большие тайм-боксы: фиксированной продолжительности, но с возможностью добавить время для завершения работы. Например, сеанс средней продолжительности может длиться час плюс-минус 15 минут. Идея заключается в том, что если мы достигли часовой отметки, но все еще хотим продолжать тестирование, то можем это сделать в течение 15 минут.

---

[1] Техника работы со списком задач, когда на выполнение определенных действий закладывается фиксированный период времени, называемый тайм-боксом. — *Примеч. ред.*

При таком подходе мы можем установить для себя крайний срок, но при этом иметь некоторую свободу действий. И опять же, если мы не закончили, то всегда можем провести дополнительные сеансы.

Не по уставу

Наконец, во время сеанса мы можем обнаружить, что вышли за рамки устава, то есть исследуем области системы, которые не относятся к запланированному сеансу. Возможно, нас привлекла другая функция или мы обнаружили проблемы, которые стоит изучить. Степень отхода от устава условна, и обычно во время сеанса может выполняться небольшой объем работ вне устава. Например, во время своего сеанса я зафиксировал работу вне устава в заметках, как показано на рис. 5.8.

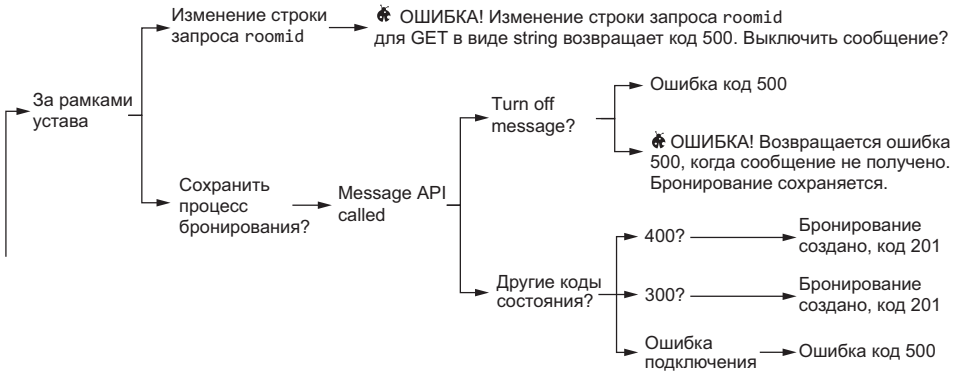

Рис. 5.8. Заметки о тестировании, описывающие действия за рамками устава

Мы не обязаны останавливаться, когда выходим за рамки устава. На самом деле мы можем просто признать, что первоначальное намерение изменилось. Но иногда может быть разумным остановиться, сделать шаг назад, подумать о новой области исследования и зафиксировать ее как новый устав.

5.3.7. Проведение собственного сеанса исследовательского тестирования

На этом мы завершаем рассмотрение примера исследовательского тестирования. Мы изучили множество аспектов этой техники, но лучший способ освоить ее — это практика. В качестве упражнения возьмите устав, который вы записали

ранее в этой главе, и проведите сеанс исследовательского тестирования на его основе. Поэкспериментируйте со следующим:

- различные эвристики для инициирования идей тестирования;
- различные инструменты для поддержки исследования API;
- разные подходы к ведению записей.

5.4. ДЕЛИТЕСЬ СВОИМИ НАХОДКАМИ

По завершении сеанса важно поделиться его результатами с коллегами. Информация, которую мы обнаружили, имеет ценность только в том случае, если команда использует новые знания в работе.

Общим подходом является подведение итогов совместно с другими членами команды. Используйте записи о тестировании, чтобы поделиться с командой следующей информацией:

- задачи сеанса;
- типы проведенного тестирования;
- обнаруженные проблемы;
- препятствия, мешавшие тестированию;
- удалось ли удержаться в рамках устава;
- необходимость дополнительных сеансов.

Это лишь несколько тем, которые можно обсудить при подведении итогов. Характер беседы позволяет подробно рассказать о сеансе, подобно тому как я поделился с вами деталями своего сеанса исследовательского тестирования.

В рамках подведения итогов также можно поразмышлять о качестве тестирования. Можно обсудить потенциальные недостатки и упущенные возможности и определить области для улучшения. Обмен опытом означает, что мы растем как тестировщики. Как будет строиться подведение итогов, зависит только от самих участников. Можно следовать структурированному списку вопросов или использовать формат неформальной беседы, на которую отводится ограниченное время. Важно, чтобы полученная информация стимулировала команду задуматься о качестве продукта и внести изменения, если это необходимо.

Упражнение

Поделитесь с одним из коллег тем, что вы обнаружили в ходе исследовательского тестирования. Поделитесь подробностями о том, что вы узнали, какие типы идей для тестирования придумали, и мыслями о качестве тестирования. Поощряйте партнера задавать вопросы, чтобы глубже осмыслить результаты тестирования. После этого поразмышляйте о том, чем вы поделились и как это помогло вам понять ценность проведенного тестирования.

5.5. ИССЛЕДОВАТЕЛЬСКОЕ ТЕСТИРОВАНИЕ КАК ЧАСТЬ СТРАТЕГИИ

Джеймс Линдсей (James Lyndsay) в работе «Исследование и стратегия» (Exploration and Strategy) предложил модель, которую мы используем для лучшего понимания нашей стратегии тестирования. Линдсей с ее помощью объяснил, как автоматизированное и исследовательское тестирование основывается на разных выявленных рисках. Он также отметил, что успешная стратегия должна включать оба этих вида тестирования.

Чтобы осмыслить это, вспомните первое задание в начале главы, когда мы следовали сценарию, исследовали пространство, а затем сравнивали результаты. Первое задание на следование инструкциям имитирует автоматизированный тест и заставляет человека сосредоточиться на конкретных инструкциях, которые вытекают из его ожиданий и знаний. Если представить это в контексте нашей модели тестирования из главы 1, то автоматизированные тесты всегда фокусируются только на совпадении представлений (воображения) и реализации (рис. 5.9).

Автоматизированные тесты требуют формулировки ожиданий от системы. Общей основой для автоматизированных тестов являются требования, и именно поэтому они фокусируются на области совпадения воображения и реализации, поскольку реализация происходит из тех же требований. Например, если требование гласит, что поле формы должно принимать действительные телефонные номера, то именно таким оно и будет создано, а автоматизированный тест будет выполнять проверку обработки действительных телефонных номеров, не более и не менее. То же самое происходит в первой части нашей работы — человек фокусируется на наших указаниях, игнорируя другие детали.

Проблема этого подхода, как видно из рис. 5.9, в том, что он оставляет много непроверенного. Мы, возможно, захотим проверить ввод в поле номера телефона других форматов номеров, недопустимых цифр, букв и символов, разного

Рис. 5.9. Фокус автоматизированных тестов в рамках стратегии тестирования

количества символов, повторных значений и т. д. Как показано на рис. 5.10, продукты обычно имеют такой уровень сложности, что мы должны исследовать их, выходя за рамки наших ожиданий. Только так можно узнать, что на самом деле происходит внутри.

Рис. 5.10. Исследовательское тестирование сосредоточивается на стороне реализации модели стратегии тестирования

В этой главе мы узнали, как исследовательское тестирование помогает расширить понимание работы приложения, выходя за пределы наших ожиданий. Таким образом, хорошая стратегия может получить пользу как от автоматизированного, так и от исследовательского тестирований. Как мы увидим в следующей

главе, сценарии тестирования могут быть отличными кандидатами для автоматизации, поскольку сценарии, как правило, алгоритмичны по своей природе. Но для истинной оценки того, как работает приложение, надо давать команде возможность целенаправленного исследования. При этом не следует забывать, что это исследование должно быть разумно структурировано, чтобы принести максимум пользы.

ИТОГИ

- Уставы — это короткие и конкретные фразы, которые помогают направлять исследовательское тестирование.
- Уставы — это заявления о намерении исследовать те области системы, которые содержат потенциальные риски.
- Для формирования уставов можно использовать различные типы шаблонов.
- Уставы выполняются в сеансах исследовательского тестирования. По одному уставу могут быть выполнены несколько сеансов тестирования.
- Уставы помогают организовать исследовательское тестирование и фиксировать его результаты.
- Эвристика помогает генерировать идеи для тестирования как подсознательно, так и осознанно.
- Оракулы — это способ определить, являются ли наблюдаемые эффекты проблемами. Существует множество разных типов оракулов, которые можно использовать.
- Для расширения возможностей исследовательского тестирования применяется дополнительное программное обеспечение.
- Ведение заметок — важный аспект исследовательского тестирования. Существует множество подходов к ведению заметок.
- Наблюдайте за своим состоянием во время тестирования. Если у вас заканчивается энергия или идеи, возможно, пришло время остановиться.
- Очень важно поделиться результатами сеанса исследовательского тестирования с коллегами.

6 Автоматизация тестирования веб-интерфейса API

В этой главе

- ✓ Что может и чего не может сделать автоматизация
- ✓ Риски, которые автоматизация поможет снизить
- ✓ Как настроить автоматизацию для тестирования API
- ✓ Как создать набор автоматизированных проверок API

По данным компании Global Market Insights, в 2019 году объем рынка автоматизированного тестирования составлял 19 млрд долларов США. По прогнозам, он может вырасти до 36 млрд долларов США к 2026 году, что отражает рост спроса на автоматизированное тестирование. Поскольку от команд ожидают быстрого создания более сложных продуктов, растет желание использовать автотестирование в качестве основной стратегии. К сожалению, эта золотая лихорадка сопровождается множеством ошибочных представлений о возможностях автоматизации, что создает неверные ожидания относительно качества автоматически протестированных продуктов.

Однако проблема заключается не в самих инструментах, а в восприятии того, как их можно использовать. Когда автоматизация используется правильно,

она может стать важным активом в стратегии тестирования. Но для этого необходимо знать ограничения и преимущества использования автоматизации в стратегии тестирования, а также понимать, как ее правильно реализовать. Поэтому в этой главе мы начнем с обсуждения того, как извлечь максимальную пользу из автоматизации, а затем рассмотрим, как реализовать ее в нашей стратегии тестирования.

6.1. ПОЛУЧЕНИЕ ПОЛЬЗЫ ОТ АВТОМАТИЗАЦИИ

Как мы узнали в предыдущей главе, посвященной исследовательскому тестированию, автоматизация может применяться по-разному. Но когда автоматизация обсуждается в контексте тестирования, обычно имеется в виду *автоматизированное регрессионное тестирование* — техника, при которой набор тестов выполняется машиной, а результаты каждого теста сообщаются как успех или провал. Это позволяет обнаружить потенциальные проблемы, которые могут ухудшить качество нашего продукта. Рассмотрим две задачи:

1. Выбор правильных объектов для автоматизации.
2. Внедрение автоматизации таким образом, чтобы она была надежной и простой в обслуживании.

Начнем с выбора правильных объектов для автоматизации. Не имеет значения, насколько хорош ваш код, если вы автоматизируете неправильные вещи.

6.1.1. Иллюзии автоматизации

Привлекательность автоматизации тестирования может быть очень высокой. Использование машин для выполнения тестирования за нас вызывает в воображении идеи более быстрой разработки, повышения производительности и сокращения затрат на тестирование вручную (недавно я видел шуточный вебинар под названием «Вечеринка по случаю выхода на пенсию отдела QA[1]», по крайней мере, я надеюсь, что он был шуточным!), но реальность может быть совсем другой. Автоматизация требует больших инвестиций. Она может замедлить работу команд, если будет ненадежной, и что хуже всего, она может ввести команды в заблуждение относительно качества продукта, который они создают.

[1] QA (quality assurance) — обеспечение качества при разработке программного обеспечения. — *Примеч. ред.*

Это не означает, что автоматизация не имеет смысла. Но слишком часто в дискуссиях предполагается, что тестирование, которое выполняет человек, равнозначно тестированию, выполняемому инструментом. Для примера представим, что мы создали автоматизированный тест, который должен проверить главную страницу сайта (это отступление от API, но оно помогает проиллюстрировать суть проблемы). На странице есть объявление, показывающее следующее учебное мероприятие, на которое можно записаться (рис. 6.1).

Автоматизированный тест открывает браузер, загружает главную страницу и находит в браузере элемент, имеющий класс `next-training-label`. Если элемент существует и отображается, то автоматизированный тест проходит успешно. Теперь представим, что мы снова запустили автоматизированный тест, но на этот раз браузер отображает страницу по-другому, как показано на рис. 6.2.

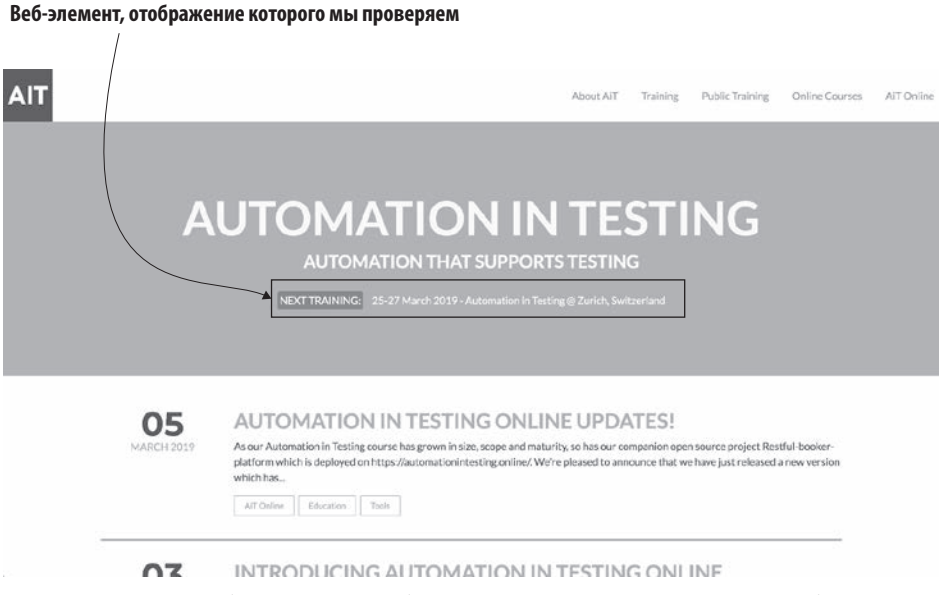

Рис. 6.1. Пример веб-страницы с рабочим CSS, показывающим детали события, которые мы хотим проверить

Сравнивая рис. 6.2 с рис. 6.1, мы интуитивно понимаем: что-то не так, качество оформления главной страницы не соответствует требованиям. Кто захочет покупать тренинг у человека с таким сайтом? Однако автоматизированный тест все равно пройдет успешно (строго говоря, для таких случаев существуют инструменты визуального тестирования, но смысл примера — демонстрация ограниченных возможностей автоматизированных инструментов).

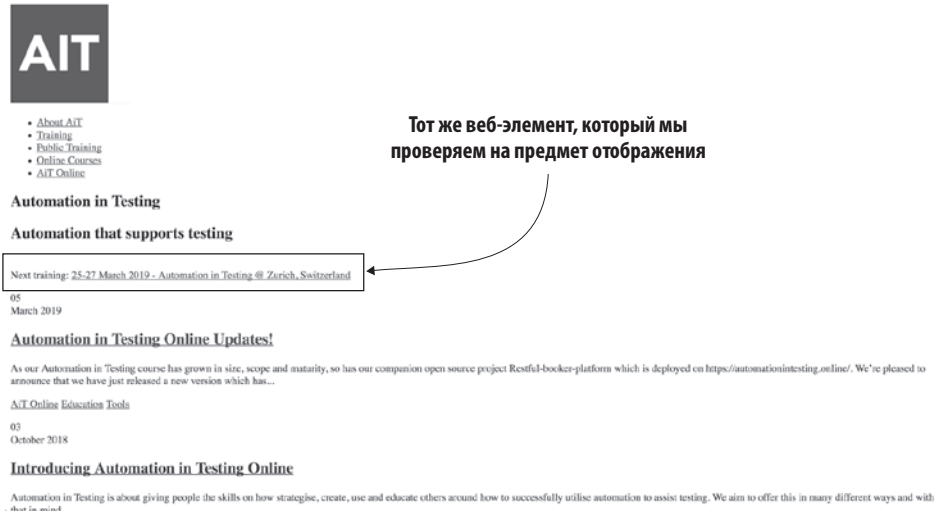

Рис. 6.2. Пример веб-страницы со сломанным CSS. Но подробности события все еще отображаются

Означает ли это, что человек всегда лучше машины? Не совсем. Средства автоматизации очень хороши для быстрого получения обратной связи в последовательной форме на основе явных инструкций, которые мы задаем. Но мы получаем то, что вкладываем, не более того. Профессиональное тестирование, проводимое человеком, может быть более медленным и трудным для повторного воспроизведения. Но мы, люди, замечательно умеем находить закономерности, и наша способность одновременно замечать множество вещей как сознательно, так и бессознательно делает нас такими хорошими тестировщиками. Автоматика скажет нам только то, о чем мы ее попросили, и ничего больше.

Проверка и тестирование

Тестирование, проводимое автоматикой, некоторые специалисты (включая меня) предпочитают называть «проверкой». Машина предоставляет обратную связь на основе заданных нами явных шагов, то есть выполняет проверку, в то время как человек, использующий эвристику, предубеждения, оракулы и многое другое, занимается тестированием. В дальнейшем я буду использовать термины «проверка» и «тестирование», чтобы различать эти средства. Хотя в конечном итоге не имеет значения, как мы это называем, пока понимаем разницу между тем, что могут внести в стратегию тестирования человек и машина.

Проблема возникает, когда команда выбирает единую стратегию тестирования, которая основана исключительно на автоматизации как можно большего количества тестов. Хотя автотесты проходят быстрее, богатство обратной связи и наблюдений теряется. Выбор в пользу автоматики жертвует качеством информации ради скорости. Опять же, нельзя делать вывод, что автоматизация — это плохо. Если мы осознаем, что инструменты по своей природе предоставляют обратную связь только о том, что мы явно просим их сообщать, то мы можем использовать автоматизацию в своих интересах. Можно использовать автоматизацию в сочетании с другими видами тестирования, чтобы добиться баланса между качеством получаемой информации и скоростью проведения тестов.

6.1.2. Автоматизация для выявления изменений

В 2012 году Майкл Болтон (Michael Bolton) на вебинаре *«Things Could Get Worse: Ideas About Regression Testing»* («Все может стать еще хуже: идеи о регрессионном тестировании») обозначил интересную проблему. Как правило, регрессионное тестирование (независимо от того, автоматизировано оно или нет) рассматривается как средство поиска проблем, которые приводят к ухудшению качества (иными словами, к регрессии) наших продуктов. Однако если качество — это нечто изменчивое, определяемое нашими пользователями, их желаниями и потребностями, то разве успех или провал на этапе регрессионного тестирования действительно говорят нам об изменении качества? Разве это не должен решать пользователь?

Майкл утверждает, что тестирование или автоматизированная проверка, которые мы проводим, не определяют, был ли спад в качестве, — это лишь наша интерпретация результатов. Если мы применим это мышление к автоматизированной регрессионной проверке, то лучше поймем симбиотические отношения между нами и нашей автоматикой. Автоматизированные проверки действуют как детекторы изменений (так их называет Майкл), а при обнаружении изменений мы определяем, вызвали ли они снижение качества.

Понимание автоматизированных проверок как детекторов изменений важно, поскольку оно помогает решить, что именно мы хотим автоматизировать. Вместо того чтобы пытаться автоматизировать тестирование всех путей и конечных точек API, наша цель — использовать автоматизированные проверки как индикаторы того, что в системе что-то изменилось. В идеале эти индикаторы следует разместить в наиболее важных областях системы. Это поможет сосредоточиться на том, что автоматизировать, но как определить наиболее важные для нас области?

6.1.3. Пусть риск будет нашим проводником

Подход к автоматизированным проверкам как к детекторам изменений помогает сформулировать представление о том, что мы хотим проверить. Мы можем построить модель покрытия, которая фокусируется на предоставлении обратной связи по тем областям, которые нас больше всего волнуют. Вместо того чтобы следить за покрытием кода и удовлетворением требований либо автоматизировать старые тестовые случаи, мы можем руководствоваться рисками, определяя, что автоматизировать в первую очередь. Например, для `booking` API мы можем определить следующие риски:

- *Невозможно оформить бронирование* — мы создаем проверку, которая отправит запрос и определит, получим ли мы положительный ответ.
- *Данные бронирования обрабатываются неправильно* — мы создаем проверку, которая посылает заказ со значениями, которые должны быть действительными, и контролирует, приходит положительный ответ или ошибка.
- *Неправильный код состояния при удалении бронирования* — мы создаем проверку, которая создает бронирование с правильными учетными данными, затем удаляет его и подтверждает, что возвращается правильный код состояния.

Как только эти риски определены, мы можем начать расставлять их в порядке приоритетности в зависимости от того, что нас больше беспокоит. Например, мы можем посчитать, что непоследовательная обратная связь — это риск, о котором можно беспокоиться меньше, чем о невозможности создать комнату. Затем порядок рисков (и предлагаемых решений) становится списком дел: что необходимо автоматизировать. Следуя этому списку, мы определяем и автоматизируем те детекторы изменений, которые наиболее важны для нас. Это гарантирует, что автоматизация будет иметь ценность.

Модульная проверка и TaTTa

Хотя мы рассмотрим, как можно создавать автоматизированные проверки API, было бы глупо не обсудить модульную проверку как часть стратегии тестирования API. Автоматические проверки API, безусловно, полезны, но если есть возможность перенести их на более низкий уровень, то есть сделать модульными, то надо это сделать. Один из методов, который я разработал, чтобы определить, следует ли что-то писать на уровне API или на уровне модулей, заключается в том, чтобы спросить себя, *тестирую ли я API или через API* (testing the API or testing through the API, сокращенно TaTTa)? Например, если внимание сосредоточено

на рисках, связанных с возвратом нормальных кодов состояния при отправке неверных данных, я могу тестировать API. Однако если я тестирую расчет налогов, то, возможно, использую API как шлюз для кода внутри API, для которого, возможно, лучше применить модульную проверку.

Упражнение

Выберите одну из конечных точек `booking` API в нашей песочнице и напишите список рисков, которые, по вашему мнению, могут повлиять на эту конечную точку. Расположите риски в порядке приоритетов и опишите, как вы будете проверять, чтобы убедиться, что проблема не проявится. Мы вернемся к этому списку ниже в данной главе.

6.2. НАСТРОЙКА ИНСТРУМЕНТА АВТОМАТИЗАЦИИ WEB API

Мы уже обсудили теоретические основы успешной автоматизации, а теперь давайте применим их на практике. Рассмотрим, как настроить и внедрить автоматизированные проверки API, сосредоточившись на трех рисках:

- `GET /booking/` всегда возвращает код нормального состояния.
- `POST /booking/` всегда успешно создает бронирование.
- `DELETE /booking/{id}` всегда удаляет бронирование.

Первым шагом будет установка зависимостей, необходимых для создания фреймворка. Затем разберемся, как мы можем структурировать наш фреймворк, чтобы создать полезную автоматизацию.

Модель проверки API

В этой главе мы будем разрабатывать проверки API на языке Java. Однако основные принципы организации такой проверки могут быть описаны на любом языке. Рекомендую вам в любом случае ознакомиться с тем, как мы автоматизируем проверки API, чтобы их было легко читать и поддерживать. Кроме того, вы можете перейти по ссылке https://github.com/mwinteringham/api-framework/ и посмотреть, как этот шаблон используется в разных языках.

6.2.1. Зависимости

Мы будем использовать фреймворк Maven для работы с зависимостями, поэтому начнем с создания нового проекта Maven и добавления в файл POM.xml следующих зависимостей, составляющих наш фреймворк:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.7.1</version>
    </dependency>
    <dependency>
        <groupId>io.rest-assured</groupId>
        <artifactId>rest-assured</artifactId>
        <version>4.3.3</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.12.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.datatype</groupId>
        <artifactId>jackson-datatype-jsr310</artifactId>
        <version>2.11.4</version>
    </dependency>
</dependencies>
```

После добавления и импорта этих зависимостей у нас имеется все необходимое для создания базовой системы автоматизации API. Давайте проведем краткий экскурс по каждой из зависимостей, чтобы лучше понять, какой вклад они вносят в общую систему.

Junit

Если вы выполняли проверки модулей в Java, то, скорее всего, знакомы с JUnit. По сути, JUnit служит инструментом, который организует и выполняет наш код.

В частности, мы будем использовать аннотацию `@Test` для организации проверок, а также встроенную зависимость `assertion` для проверки ответов.

REST assured

REST Assured отвечает за создание и выполнение HTTP-запросов и разбор HTTP-ответов. Мы можем рассматривать его как «движок» фреймворка. Существует множество зависимостей Java, которые позволяют создавать HTTP-запросы и анализировать ответы. Мы выбрали REST Assured, но с учетом структуры нашего фреймворка достаточно легко заменить его другим инструментом, например Spring или java.net.http, без ущерба для автоматических проверок.

Тем не менее мы будем использовать REST Assured из-за его простоты и ясности при создании запросов. Вы можете узнать больше о REST Assured на сайте https://rest-assured.io/.

Jackson

Зависимости `databind`, `core` и `datatype-jsr310` помогают преобразовывать старый добрый Java-объект POJO в JSON для наших HTTP-запросов, и наоборот — JSON в HTTP-ответах в пригодные для использования объекты.

> **А другие зависимости?**
>
> Пять зависимостей обеспечивают все необходимое для создания базовой системы автоматизации API. В дальнейшем вы, возможно, захотите расширить вашу систему: улучшить отчетность из фреймворка, расширить подход к подтверждениям для обработки больших тел ответов или сократить кодовую базу. Все эти возможности доступны, но всему свой черед.

6.2.2. Структурирование фреймворка

Верный способ потерпеть неудачу в автоматизации — это создать структуру, которую трудно поддерживать или читать. Важно помнить, что код автоматизации — это именно код. Мы должны использовать для него те же методы и подходы, которые применяются для поддержания чистоты, читабельности и удобства сопровождения рабочего кода. После реализации большого количества автоматизаций API и изучения специальных моделей (например, Page Object в UI-автоматизации) я пришел к выводу, что наиболее эффективной является структура фреймворка, показанная на рис. 6.3.

Давайте кратко разберем каждую из этих областей, чтобы объяснить их роль в системе.

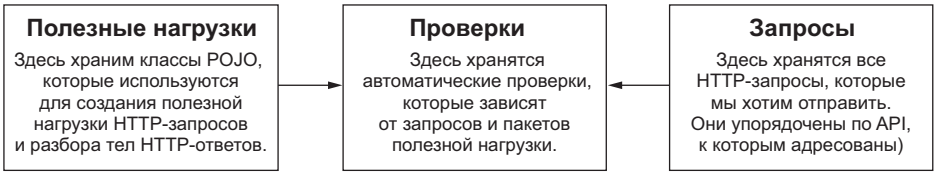

Рис. 6.3. Модель, показывающая взаимосвязи трех основных областей структуры проверки API

Проверки

Здесь мы создаем и организуем проверки, как это делается в любой системе автоматизации. Цель этой части системы — сделать автоматизированные проверки максимально простыми для восприятия. Понимание замысла автоматизированной проверки очень важно. Если вы не знаете, что она делает, как вы узнаете, действительно ли она прошла успешно?

Генерация данных, отправка HTTP-запросов и прием ответов выполняются в других разделах фреймворка. Если мы дадим понятные имена методам, объектам и утверждениям, коллегам будет проще разобраться, на что направлена проверка.

Исходя из этого, создадим новый пакет под названием `com.example.checks`, в котором будем хранить наши автоматизированные проверки.

Запросы

Как правило, разработчики фреймворков для автоматизации API не абстрагируют HTTP-запросы в отдельную область. Это вызывает всевозможные сложности по мере роста числа автоматизированных проверок. Например, обновление URL в тысячах проверок — крайне неэффективное использование времени.

Итак, в пакете запросов мы организуем наши классы в соответствии с веб-API, к которому обращаемся. Каждый запрос будет иметь свои собственные методы, которые мы затем вызываем в области проверок фреймворка. Таким образом, если нам нужно обновить URL, то нужно изменить его только в одном месте.

Для этой части фреймворка мы создадим дополнительный пакет `com.example.requests`, в который будем добавлять классы запросов по мере создания автоматизированных тестов.

Полезные нагрузки

Многие веб-API используют сложные модели запросов и ответов, что приводит к появлению большого количества классов POJO. Целью данной области

фреймворка является эффективная организация этих классов в зависимости от того, применяются они для запросов или ответов, а также разделение их по отдельным областям в зависимости от того, к какому API вы обращаетесь.

Для этой части фреймворка создадим пакет `com.example.payloads`. Затем можно расположить POJO-классы внутри пакета по их API, если это необходимо.

После завершения экскурса по зависимостям и структуре фреймворка у нас должен получиться проект, который выглядит примерно так, как показано на рис. 6.4.

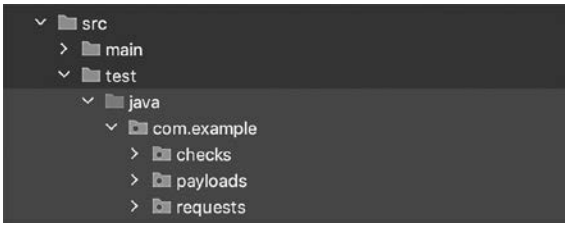

Рис. 6.4. Схема проекта фреймворка для проверки API, взятая из IntelliJ

6.3. СОЗДАНИЕ АВТОМАТИЗИРОВАННЫХ ПРОВЕРОК API

Теперь, когда все готово, давайте приступим к созданию автоматических проверок для наших трех рисков.

6.3.1. Автоматическая проверка 1: GET-запрос

Для начала мы создадим проверку, которая отправляет GET-запрос на URL https://automationintesting.online/booking/ и подтверждает, что в ответ мы получаем код состояния `200`.

Создание пустой основы проверки

Сначала нам нужно создать класс, в котором будут храниться проверки. В пакете `com.example.checks` создайте класс с именем `BookingApiIT`. После этого мы напишем нашу первую пустую проверку следующим образом:

```
public class BookingApiIT {

  @Test
  public void getBookingSummaryShouldReturn200(){

  }
}
```

Здесь аннотация `@Test` сообщает JUnit (а также вашей IDE), что метод `getBookingShouldReturn200` — это проверка, которую можно запустить. Это дает основу для запуска проверки.

Создание get-запроса

После создания проверки добавим возможность отправлять и получать HTTP-запросы и ответы. Начнем с создания нового класса в `com.example.requests`, который будет называться `BookingApi`.

Однако прежде чем добавить в класс какой-либо код, давайте на секунду прервемся и коротко остановимся на том, как именовать классы в пакете запросов. Соглашение об именовании зависит от того, отвечает код за тестирование одного или нескольких веб-API. Правило, которого следует придерживаться, выглядит следующим образом:

- *Если тестируем только один API*, называйте классы в зависимости от ресурсов. Например, API Ghost CMS — это один API с множеством ресурсов (https://ghost.org/docs/content-api/), поэтому мы можем организовать классы по ресурсам, таким как посты, страницы и теги.
- *Если тестируем несколько API*, называйте классы по имени каждого веб-API. Например, на платформе restful-booker есть такие API, как `booking`, `auth`, `room` и т. д. Каждый из них будет иметь свой собственный класс.

Я считаю, что такое именование помогает хорошо разделять запросы и делает код более читабельным.

Создав класс `BookingApi`, мы добавим следующий код:

```
public class BookingApi {

    private static final String apiUrl =
    ➥"https://automationintesting.online/booking/";   <-- Объявляет базовый URL

    public static Response getBookingSummary(){   <-- Отправляет GET-запрос
        return given().get(apiUrl + "summary?roomid=1");
    }
}
```

`apiUrl` гарантирует, что если в будущем потребуется заменить наш базовый URL на `/booking/`, это можно будет сделать в одном месте. Или, если мы работаем с несколькими окружениями, это можно будет контролировать с помощью переменных окружения.

Метод `getBookingSummary` содержит код REST Assured для отправки запроса. Как вы можете видеть, метод `get()` выполняет большую часть работы за нас. Предоставив методу `get()` значение URL, REST Assured создает все необходимое для отправки базового GET-запроса к нашему веб-API. Метод `given()` фактически ничего не делает, кроме как улучшает читабельность нашего автоматизированного теста благодаря подходу *Given-When-Then*. Если вы не знакомы с этим синтаксисом, не беспокойтесь, мы будем использовать его только в случае необходимости.

Наконец, когда запрос будет завершен, REST Assured ответит объектом `Response`, который мы проверим, чтобы узнать, как веб-API ответил на запрос.

Создав метод `getBookingSummary`, добавим его в нашу проверку вместе с подтверждением следующим образом:

```
@Test
public void getBookingSummaryShouldReturn200(){
    Response response = BookingApi.getBookingSummary();

    assertEquals(200, response.getStatusCode());
}
```

Теперь наша проверка использует `getBookingSummary` для отправки GET-запроса на `/booking/` и получает объект `Response`, из которого мы извлекаем код состояния с помощью `getStatusCode`, чтобы проверить, является ли он кодом состояния `200`.

Если мы запустим проверку сейчас, то увидим, что она проходит и возвращает правильную информацию. Можно изменить код состояния на другое число и посмотреть, как автоматическая проверка провалится.

6.3.2. Автоматическая проверка 2: POST-запрос

Следующая автоматизированная проверка проконтролирует, что POST-запрос на https://automationintesting.online/booking/ с допустимой полезной нагрузкой возвращает код состояния `201`.

Создание пустой основы проверки

Начнем с добавления новой проверки в класс `BookingApiIT` следующим образом:

```
@Test
public void postBookingReturns201(){

}
```

Создание pojo

Перед отправкой POST-запроса необходимо создать полезную нагрузку `Booking`. Вот пример полезной нагрузки в формате JSON:

```
{
    "roomid": Int
    "firstname": String,
    "lastname": String,
    "depositpaid": Boolean,
    "bookingdates": {
        "checkin": Date,
        "checkout": Date
    },
    "additionalneeds": String
}
```

Для получения полезной нагрузки создадим класс POJO, который структурирован так же, как наш объект JSON, а затем отправим экземпляр этого POJO в библиотеку запросов.

Сначала создадим два новых класса в пакете `com.example.payloads` под именами `Booking` и `BookingDates`. Затем откроем класс `BookingDates` и добавим в него следующее:

```
public class BookingDates {

    @JsonProperty  <-- Объявляет переменную
    private LocalDate checkin;  <-- Объявляет переменную как JsonProperty
    @JsonProperty
    private LocalDate checkout;

    public BookingDates() {}  <-- Конструктор по умолчанию, необходимый Jackson

    public BookingDates(LocalDate checkin, LocalDate checkout){
        this.checkin = checkin;
        this.checkout = checkout;
    }  <-- Пользовательский конструктор для создания полезной нагрузки

    public LocalDate getCheckin() {
        return checkin;
    }  <-- Геттер для использования Jackson

    public LocalDate getCheckout() {
        return checkout;
    }
}
```

Давайте разберемся в этом коде.

1. Мы объявляем переменные, следя за тем, чтобы имя каждой из них совпадало с именем ключа в объекте JSON. Например, мы видим, что дата регистрации в объекте `Booking` обозначена как `checkin`, поэтому мы так и называем соответствующую переменную. Мы также должны убедиться, что тип данных переменной соответствует типу данных, который будет использоваться в объекте JSON.
2. Аннотация `@JsonProperty` указывает зависимостям Jackson, какие переменные должны быть преобразованы в пары ключ — значение, когда класс POJO преобразуется в объект JSON для нашего запроса.
3. Мы создаем два конструктора: первый позволяет присваивать значения переменным, второй — пустой, о нем мы узнаем, когда будем создавать следующую автоматическую проверку.
4. Наконец, мы создаем геттеры, которые Jackson будет использовать для получения значений, присвоенных каждой переменной, чтобы добавить их в объект JSON.

Создав подобъект `BookingDates`, мы добавим следующее в класс `Booking`:

```
@JsonIgnoreType
public class Booking {

    @JsonProperty
    private int roomid;
    @JsonProperty
    private String firstname;
    @JsonProperty
    private String lastname;
    @JsonProperty
    private boolean depositpaid;
    @JsonProperty
    private BookingDates bookingdates;
    @JsonProperty
    private String additionalneeds;

    // default constructor required by Jackson
    public Booking() {}

    public Booking(int roomid, String firstname, String lastname, boolean
     depositpaid, BookingDates bookingdates, String additionalneeds) {
        this.roomid = roomid;
        this.firstname = firstname;
        this.lastname = lastname;
```

```
        this.depositpaid = depositpaid;
        this.bookingdates = bookingdates;
        this.additionalneeds = additionalneeds;
    }

    public int getRoomid() {
        return roomid;
    }

    public String getFirstname() {
        return firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public boolean isDepositpaid() {
        return depositpaid;
    }

    public BookingDates getBookingdates() {
        return bookingdates;
    }

    public String getAdditionalneeds() {
        return additionalneeds;
    }
}
```

Как видим, схема такая же: создание переменных, соответствующих ключам объекта JSON, добавление `@JsonProperty` к каждой из них и создание необходимых конструкторов и геттеров. Мы также добавили `@JsonIgnoreType`, который будет использоваться в следующей проверке, но пока игнорируйте его. Кроме того, обратите внимание, как мы используем класс `BookingDates` для подобъекта `BookingDates`. Вот так мы создаем сложные полезные нагрузки JSON путем задания связей между классами, которые соответствуют связям между JSON-объектами.

Теперь мы можем создать полезную нагрузку, которую хотим отправить в запросе, обновив проверку следующим образом:

```
@Test
public void postBookingReturns201(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 1 , 1 ),
        LocalDate.of( 2021 , 1 , 3 )
    );

    Booking payload = new Booking(
        1,
```

```
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );
}
```

Мы получили все необходимое для создания полезной нагрузки с помощью Java. Равным образом можно было использовать инструменты вроде Lombok, упростив написание шаблонного кода в классах POJO. Или можно было применить шаблоны построения данных, чтобы использовать более абстрактный подход к созданию классов POJO.

Переиспользование POJO из продакшен-кода

Классы POJO широко применяются в продакшен-коде веб-API, в связи с чем возникает вопрос: почему бы не переиспользовать уже созданные классы, а не дублировать код? На этот вопрос нет однозначного ответа, но есть альтернатива. Да, переиспользование классов POJO из продакшен-кода действительно экономит время и сокращает расходы на их поддержание в актуальном состоянии. Однако при этом возникает риск ложных срабатываний в тестах. Хотя это и редкость, при переиспользовании продакшен-кода есть риск внести в объекты POJO ошибки, которые не будут отмечены при выполнении тестов. В конце концов, если вы случайно опустили важную часть класса POJO или неправильно назвали его, использование той же модели для получения тестовой полезной нагрузки приведет к созданию класса POJO, который будет принят.

Создание запроса

Теперь, когда у нас есть полезная нагрузка, мы можем создать наш POST-запрос в классе `BookingApi`, который выглядит следующим образом:

```
public static Response postBooking(Booking payload) {
    return given()
            .contentType(ContentType.JSON)  <-- Объявляет заголовок Content-Type
            .body(payload)  <-- Добавляет в тело запроса
            .when()
            .post(apiUrl);
}
```

Мы расширили использование REST Assured, предоставив заголовок содержимого с помощью `contentType()`, а также передав нашу полезную нагрузку с помощью метода `body()`. После завершения запроса возвращается объект `Response`.

Когда код запроса готов, мы можем обновить проверку, чтобы сделать запрос следующим образом:

```
@Test
public void postBookingReturns201(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 1 , 1 ),
        LocalDate.of( 2021 , 1 , 3 )
    );

    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

Response response = BookingApi.postBooking(payload);
}
```

Подтверждение ответа

Все, что осталось сделать, — добавить подтверждение в конце, чтобы проверить, возвращается ли код состояния `201`:

```
@Test
public void postBookingReturns201(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 1 , 1 ),
        LocalDate.of( 2021 , 1 , 3 )
    );

    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
);

    Response response = BookingApi.postBooking(payload);

    assertEquals(201, response.getStatusCode());
}
```

Конечно, мы можем проверить другие параметры ответа от нашего веб-API, например, содержит ли тело ответа ожидаемые значения. Однако это потребует дополнительной работы и будет рассмотрено в последней автоматизированной проверке.

6.3.3. Автоматизированная проверка 3: объединение запросов

Для последней проверки мы рассмотрим удаление бронирования, что требует нескольких шагов:

1. Создать бронирование.
2. Получить токен аутентификации, позволяющий удалить бронирование.
3. Использовать токен и идентификатор бронирования для удаления бронирования.

Нам понадобится возможность не только создавать полезную нагрузку запроса, но и анализировать тело ответа для дальнейшего использования.

Создание первоначальной проверки

Давайте начнем с создания новой проверки внутри `BookingApiIT` следующим образом:

```
@Test
public void deleteBookingReturns202(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 2 , 1 ),
        LocalDate.of( 2021 , 2 , 3 )
    );

    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

}
```

На первом шаге нам нужно создать бронирование, которое впоследствии будет удалено. Мы создали проверку и добавили начальную работу по созданию полезной нагрузки для запроса бронирования, которая использует POJO и методы запроса, созданные нами в предыдущей проверке.

Создание pojo для анализа ответа

Чтобы удалить бронирование, нам понадобится ID бронирования, который передается в запросе на удаление. Этот идентификатор возвращается в теле ответа для успешно созданного бронирования:

```
{
    "bookingid": 1,
    "booking": {
        "roomid": 1
        "firstname": "Jim",
        "lastname": "Brown",
        "depositpaid": true,
        "bookingdates": {
            "checkin": "2018-01-01",
            "checkout": "2019-01-01"
        },
        "additionalneeds": "Breakfast"
    }
}
```

Это означает, что нам придется разобрать тело ответа из JSON в POJO, к которому мы затем сделаем запрос для получения идентификатора бронирования. К счастью, большая часть работы уже выполнена, потому что объект `booking` — это та же модель, которую мы использовали для создания полезной нагрузки запроса. То есть нам нужно только создать новый класс `BookingResponse` в `com.example.payloads` и добавить следующее:

```
public class BookingResponse {

    @JsonProperty
    private int bookingid;
    @JsonProperty
    private Booking booking;

    public int getBookingid() {
        return bookingid;
    }

    public Booking getBooking() {
        return booking;
    }
}
```

Как и в других вариантах классов POJO, мы создаем переменные, соответствующие ключам объекта JSON, и присваиваем им правильные типы данных. Видите, как мы используем для этого класс `Booking`? Мы также добавляем геттеры, чтобы позже можно было извлекать нужные значения.

С `BookingResponse` объект POJO закончен, теперь мы можем обновить тест следующим образом:

```
@Test
public void deleteBookingReturns202(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 2 , 1 ),
        LocalDate.of( 2021 , 2 , 3 )
    );

    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

    Response bookingResponse = BookingApi.postBooking(payload);
    BookingResponse createdBookingResponse =
      bookingResponse.as(BookingResponse.class);
}
```

Обратите внимание на последнюю строку кода. Мы берем объект `Response` из `postBooking` и вызываем метод `.as()`, предоставляя ему структуру класса `BookingResponse`. По сути, мы просим REST Assured сопоставить значения JSON-тела ответа с классом `BookingResponse` и создать новый объект `BookingResponse`. Если POJO и JSON-тело ответа совпадают, значения из записей в JSON будут сохранены в переменных нашего объекта для последующего использования.

После этого у нас есть объект с данными ответа бронирования, необходимыми для дальнейшего удаления в процессе проверки.

Для чего нужны пустые конструкторы в Booking и BookingDates?

Вы, вероятно, помните, что ранее нам пришлось добавить пустые конструкторы для классов `Booking` и `BookingDates`, но мы не добавили их для `BookingResponse`.

Почему? Когда мы получаем HTTP-ответ с телом JSON для разбора, он возвращается как обычная строка, поэтому нам нужно преобразовать этот JSON в объект, который мы можем использовать. Для этого применяются Jackson и REST Assured, выполняющие следующие шаги:

- Создать новый пустой объект на основе предоставленного класса (в нашем примере это `BookingResponse.class`).
- Выполнить перебор ключей в объекте JSON и, используя метаданные, взятые из `@JsonProperty` над каждой переменной, найти в объекте нужную переменную для хранения соответствующего значения.

Для успешного осуществления этого процесса нужна возможность создать пустой объект, что Jackson может сделать за нас автоматически, если мы не будем добавлять пользовательские конструкторы. Именно поэтому `Booking` и `BookingDates` требуют явного создания пустых конструкторов. Если бы они отсутствовали, Jackson и REST Assured попытались бы использовать пользовательский конструктор, содержащий параметры, и в конечном итоге потерпели бы неудачу.

Один из способов разобраться в этой концепции — зайти в `Booking` или `BookingDates`, попытаться удалить пустой конструктор и запустить проверку. Это, скорее всего, приведет к ошибке, подобной следующей:

```
com.fasterxml.jackson.databind.exc.InvalidDefinitionException: Cannot
construct instance of `com.example.payloads.Booking` (no Creators, like
default constructor, exist)....
```

Главное, что нужно помнить: если вы увидите подобную ошибку в будущем, значит, скорее всего, вам нужно явно создать пустой конструктор в вашем объекте POJO.

Сделайте то же самое для auth

Мы создали бронирование, которое в конечном итоге удалим, но нам все еще нужно авторизоваться, чтобы запрос на удаление был обработан. Для этого понадобится полезная нагрузка авторизации, позволяющая создавать и получать токены, которые мы отправляем и получаем от `auth` API. Нам нужно создать два новых класса POJO в `com.example.payloads`. Первый, `Auth`, который мы будем использовать для создания запросов к полезной нагрузке, выглядит следующим образом:

```
public class Auth {

    @JsonProperty
    private String username;
    @JsonProperty
    private String password;

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public Auth(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

Создание запросов

Когда объекты POJO созданы, следующим шагом будет написание кода для отправки запроса на `/auth/login`. Поскольку мы еще не отправляли запросы к этому API, создадим новый класс `AuthApi`. Однако перед этим надо перенести некоторые детали из `BookingApi` в собственный класс, который может быть общим для всех наших классов `*Request`.

В `com.example.requests` создадим новый класс `BaseApi` и добавим в него строки:

```
public class BaseApi {

    protected static final String baseUrl =
    ➥"https://automationintesting.online/";
}
```

После создания `BaseApi`, который отвечает за управление `baseUrl` всех наших запросов, мы можем обновить `BookingApi` следующим образом:

```
public class BookingApi extends BaseApi {

    private static final String apiUrl = baseUrl + "booking/";
```

Затем мы создадим класс `AuthApi` и добавим в него следующее:

```
public class AuthApi extends BaseApi {

    private static final String apiUrl = baseUrl + "auth/";

    public static Response postAuth(Auth payload){
        return given()
            .contentType(ContentType.JSON)
            .body(payload)
            .when()
            .post(apiUrl + "login");
    }
}
```

Связать все воедино и добавить подтверждение

Теперь есть все необходимое для обновления нашей автоматической проверки, чтобы получить токен аутентификации для будущих запросов, как показано ниже:

```
@Test
public void deleteBookingReturns202(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 2 , 1 ),
        LocalDate.of( 2021 , 2 , 3 )
    );

    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

    Response bookingResponse = BookingApi.postBooking(payload);
    BookingResponse createdBookingResponse =
    ➥bookingResponse.as(BookingResponse.class);
    Auth auth = new Auth("admin", "password123");

    Response authResponse = AuthApi.postAuth(auth);
        String authToken = authResponse.getCookie("token");

}
```

С помощью кода, создающего бронирование для удаления и получающего авторизационный токен, мы можем завершить автоматическую проверку для

удаления бронирования и подтвердить ответ. Вначале давайте создадим метод, который отправит запрос на удаление в `BookingApi`:

```
public static Response deleteBooking(int id, String tokenValue) {

    return given()
            .header("Cookie", "token=" + tokenValue)
            .delete(apiUrl + Integer.toString(id));
}
```

Затем мы обновляем проверку, чтобы удалить бронирование, и подтверждаем ответ с кодом состояния следующим образом:

```
@Test
public void deleteBookingReturns202(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 2 , 1 ),
        LocalDate.of( 2021 , 2 , 3 )
    );

    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

    Response bookingResponse = BookingApi.postBooking(payload);
    BookingResponse createdBookingResponse =
      bookingResponse.as(BookingResponse.class);

    Auth auth = new Auth("admin", "password");

    Response authResponse = AuthApi.postAuth(auth);
        String authToken = authResponse.getCookie("token");

    Response deleteResponse = BookingApi.deleteBooking(
            createdBookingResponse.getBookingid(),
            authToken);

    assertEquals(202, deleteResponse.getStatusCode());
}
```

На этом наша последняя автоматизированная проверка завершена. Этот пример автоматизированной проверки демонстрирует, что мы можем извлекать

значения из тел HTTP-ответов для использования в других запросах или их подтверждения.

Автоматизация GraphQL

Поскольку GraphQL обычно обслуживается по HTTP, мы можем использовать тот же набор инструментов для автоматизации GraphQL API, что и для REST API. Разница между ними заключается в том, как вы создаете свой HTTP-запрос. Например, если вы хотите отправить запрос, подобный следующему:

```
query Character {
  character(id: 1) {
    name
    created
  }
}
```

нужно создать класс POJO, который будет представлять эту структуру, и отправить его в REST Assured таким же образом, как мы отправляли полезную нагрузку в примерах выше. Чтобы узнать больше об использовании REST Assured с GraphQL, читайте блог специалистов компании Applitools (http://mng.bz/z5J1).

Упражнение

Теперь, когда создан фреймворк автоматизации, вернитесь к списку рисков и проверок, которые вы определили, выберите из него самую важную проверку и попытайтесь ее автоматизировать. Чтобы укрепить свои знания и навыки в использовании этого подхода, автоматизируйте остальные проверки из вашего списка.

6.3.4. Запуск автоматизированных тестов в качестве интеграционных

Теперь, когда у нас есть автоматизированные проверки, запустим их как часть процесса сборки. Как вы помните, мы назвали класс, содержащий наши автоматизированные проверки, `BookingApiIT`. Добавление `IT` в конце имени класса помогает отделить автоматизированные проверки от любых модульных проверок, которые мы могли создать для проверки отдельных компонентов веб-API. Это помогает четко разделить задачи и цели автоматизированных проверок. Вместе с тем, если бы мы выполнили `mvn clean install` в этом проекте прямо сейчас, наши автоматические проверки не запустились бы. Это объясняется тем, что когда `maven` проходит фазу тестирования, он ищет классы, в конце которых есть слово `Test`, и игнорирует все остальные, в том числе заканчивающиеся буквами `IT`.

Нужно обновить наш проект Maven, чтобы он распознавал автоматизированные проверки как часть процесса сборки. Для этого добавим плагин `maven-failsafe-plugin` в наш файл `pom.xml` следующим образом:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.22.2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                    <configuration>
                        <includes>
                            <include>**/*IT</include>
                        </includes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build
```

Плагин позволяет подключить запуск наших автоматических проверок к целям `integration-test` или `verify` в Maven, добавив эти цели в раздел `executions` плагина. В строке `<include>**/*IT</include>` используются подстановочные знаки для поиска любых файлов с окончанием `IT`, например нашего класса `BookingApiIT`.

Теперь команда `mvn clean verify` запустит наши автоматизированные проверки. То есть мы можем добавить наш проект автоматизированных проверок в пайплайн и запустить `mvn clean verify` на тестируемой среде. Таким образом, мы создали серию автоматизированных тестов, которые можно запускать локально или в рамках процесса непрерывной интеграции.

6.4. ИСПОЛЬЗОВАНИЕ АВТОМАТИЗАЦИИ В СТРАТЕГИИ ТЕСТИРОВАНИЯ

По мере обсуждения способов создания автоматизированных проверок становилось очевидно, что для их успешной реализации необходимы предварительные знания о системе. В отличие от других видов тестирования, которые направлены на изучение новой информации, автоматизация направлена на подтверждение

того, что мы уже знаем. Причем это ее качество остается неизменным по мере появления новых версий продукта. Если мы вернемся к модели тестирования из главы 1, то увидим, что автоматизированные проверки сосредоточены на областях, в которых наши знания о том, что мы хотим построить (воображение), и то, что мы уже построили (реализация), пересекаются, как показано на рис. 6.5.

Автоматизированные проверки — важный вид тестирования, который должен присутствовать в нашей стратегии. По мере развития продукта нужно не только тестировать новые функции и исправления, но и подтверждать, что наши ожидания относительно того, как система работала в прошлом, все еще верны. Но как мы узнали, существует разница между тем, что наши средства автоматизации сообщают о системе, и тем, что наблюдаем мы сами. Успешная стратегия балансирует между использованием автоматизации для поддержки и предупреждения о потенциальных изменениях, чтобы мы могли использовать другие виды тестирования (например, исследовательское) для получения дополнительной информации.

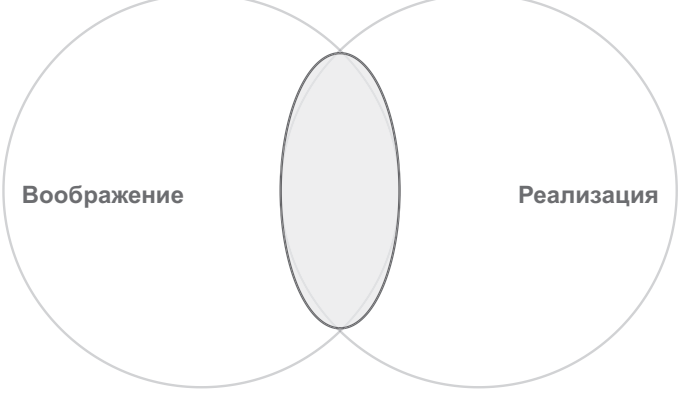

Рис. 6.5. Автоматизация сосредоточена на области совпадения воображения и реализации, то есть на том, что мы знаем о системе в настоящее время

ИТОГИ

- Автоматизация жертвует качеством обратной связи ради скорости и эффективности.
- Использование автоматизации в контексте регрессионных проверок означает создание автоматизированных проверок, которые служат в качестве детекторов изменений.

- Детекторы изменений сообщают нам об изменениях в системе.
- Мы можем определять и приоритизировать детекторы изменений на основе рисков, которые нас беспокоят.
- Мы можем создать собственные фреймворки для проверки API с помощью JUnit, REST Assured и Jackson.
- Чтобы код соответствовал принципу DRY («не повторяйся») и был удобным для сопровождения, целесообразно разделять его на три части: тесты, запросы и полезная нагрузка.
- Мы можем использовать язык и инструменты Java для создания HTTP-запросов и анализа результатов, чтобы тестировать изменения в наших API.
- Плагин maven-failsafe-plugin можно использовать для автоматических проверок API как часть пайплайна сборки.
- Мы можем использовать автоматизацию в сочетании с другими видами тестирования, чтобы эффективно поддерживать наш продукт.

7 Разработка и внедрение стратегии тестирования

В этой главе

- ✓ Как расставить приоритеты и реализовать конкретные действия как часть стратегии
- ✓ Почему для разных контекстов требуются разные стратегии
- ✓ Создание плана на основе стратегии
- ✓ Анализ рабочего контекста для создания успешного плана

Почти каждый тестировщик в какой-то момент своей карьеры осознает, что нельзя протестировать абсолютно все. Такие ограничения, как бюджет, сроки, сложность, навыки и многое другое, влияют на время, которым вы располагаете для тестирования и обучения. Времени на тестирование всегда не хватает. Чтобы решить эту проблему, мы должны стратегически подходить к выбору, что тестировать, а что нет.

Во части 2 этой книги мы рассмотрели составляющие тестирования, связанные с разными частями цикла разработки программного обеспечения и различными видами рисков, но обсуждение стратегии тестирования было абстрактным. Как определить, какие мероприятия по тестированию являются приоритетными и какие шаги необходимо предпринять для успешной реализации нашей стратегии?

То, что работает для одного проекта, может не работать для другого, поэтому в этой главе мы рассмотрим, почему не существует универсальной стратегии и какие шаги нужно предпринять, чтобы определить стратегию тестирования и приступить к ее реализации.

7.1. ОПРЕДЕЛЕНИЕ СТРАТЕГИИ ДЛЯ КОНКРЕТНОГО КОНТЕКСТА

Прежде чем перейти к рассмотрению того, как мы можем формализовать стратегию и определить шаги по ее реализации, давайте вспомним, что мы узнали до сих пор. Мы изучили две модели, связанные со стратегией тестирования. Первая — это модель цели тестирования для проверки воображаемого образа (то, что мы хотим создать) и реализации (то, что мы создали). Чем больше мы узнаем в каждой из областей, тем больше они пересекаются. Визуализация этой модели представлена на рис. 7.1.

Рис. 7.1. Модели воображаемого образа продукта и его реализации

В предыдущих главах мы рассмотрели различные виды тестирования, а также увидели, как разные техники фокусируются на разных областях модели воображаемого образа и реализации. Мы увидели, что тестирование дизайна API глубже погружает нас в область воображения, а исследовательское тестирование расширяет наши знания о том, что мы уже реализовали. Автоматизация помогает подтвердить, что то, что мы уже знаем о продукте, остается верным по мере его изменения. Все это можно обобщить, дополнив модель воображаемого образа и реализации, как показано на рис. 7.2.

Рис. 7.2. Модели воображаемого образа продукта и его реализации, дополненные мероприятиями по тестированию

Конечно, здесь представлен не полный список доступных нам видов тестирования. Но уже ясно, что разные виды выявляют различную информацию и концентрируются на разных рисках. Мы рассмотрим больше видов тестирования в части 3 этой книги, но уже сейчас понятно, как сложно определить, что и когда делать в нашей стратегии тестирования.

7.1.1. Определение приоритетных задач

Как уже говорилось, хотя и хочется сказать: «Давайте сделаем все!», в реальности почти всегда нецелесообразно стараться провести сразу все мероприятия по тестированию. Стратегия помогает определить, какие возможности доступны, какие из них должны быть приоритетными и что следует планировать в кратко- и среднесрочной перспективе. Как определить, что является приоритетным? В этом нам поможет вторая модель, о которой мы узнали, основанная на изучении качественных характеристик и рисков.

В главе 3 мы рассмотрели модель, помогающую сделать начальные шаги по выбору видов тестирования. В частности, это:

- Определение того, какие характеристики качества важны для наших пользователей и бизнеса.
- Определение рисков, которые могут повлиять на эти характеристики качества.

Это было обобщено в модели, которая демонстрирует характеристики качества и риски, как показано на рис. 7.3.

Рис. 7.3. Модель характеристик качества и рисков, показывающая, что изучение рисков является шагом к достижению цели нашей стратегии

Эта модель использовалась в главе 3 в качестве наглядной демонстрации того, как выявление каждого риска помогает команде и является шагом на пути улучшения или поддержания качества продукта. Эти же риски помогают определить, какое тестирование мы хотим провести и каков его приоритет. Например, если немного расширить аналогию с шагами, можно сказать, что первый риск в нашей модели является приоритетным:

> При успешных запросах отправляются неправильные коды состояния или ответы.

Если этот риск беспокоит нас больше всего, то приоритетным в нашей стратегии станет тестирование дизайна API.

Различные риски будут указывать на то, какие действия нам следует предпринять. Требуются умение и практика, чтобы определить, может ли риск проявиться

в результате недопонимания воображаемой части нашей модели или как неожиданный побочный эффект, обусловленный нашей реализацией. Но способность взглянуть на конкретный риск и понять, какие виды тестирования лучше всего подходят для его снижения, помогут нам выбрать наиболее эффективный подход к тестированию.

Давайте вернемся к списку рисков для нашей API-песочницы и выберем три основных риска, которые нас беспокоят:

1. При успешных запросах отправляются неправильные коды состояния или ответы.
2. Процесс проверки работает не так, как ожидалось.
3. Администратор не может обновить внешний вид и оформление гостевых страниц.

Мы можем проанализировать эти риски и определить необходимые мероприятия по тестированию, в результате чего получим модель стратегии, подобную показанной на рис. 7.4.

Рис. 7.4. Модель стратегии тестирования, содержащая приоритетные мероприятия

Обратите внимание, что мы указали приоритеты мероприятий: тестирование дизайна API обозначено как первое по приоритету, поскольку призвано снизить наиболее критический риск. Далее следует исследовательское тестирование, которое поможет снизить менее важные риски. Ориентация на качество продукта при выборе техник тестирования помогает определить и расставить приоритеты в нашей стратегии.

7.1.2. Разные стратегии для различных контекстов

Использование нашего понимания качества при тестировании может быть обобщено в модели, показанной на рис. 7.5.

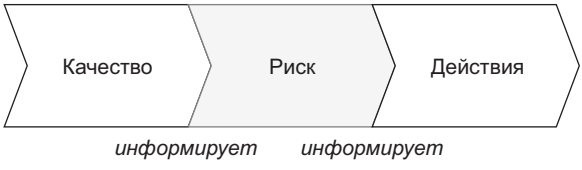

Рис. 7.5. Модель, демонстрирующая взаимосвязь между качеством, риском и тестированием

Следуя этой схеме, мы не только видим четкий путь, но и можем разработать стратегию, которая соответствует желаниям пользователей и нашему рабочему контексту.

В главе 3 вы узнали, что качество может означать разные вещи для разных людей. На качество могут влиять отрасль и сфера применения нашего продукта, например интернет вещей (IoT) или программное обеспечение как услуга (SaaS). Эти различия будут определять, что значит качество для наших пользователей и для нас.

Например, если мы рассмотрим проект, отличный от API-песочницы, такой как платформа для покупки акций и ценных бумаг, то вскоре обнаружим, что характеристики качества отличаются от нашего примера. Например, они могут включать следующие требования:

- точность;
- отзывчивость;
- контролируемость.

Эти характеристики качества связаны с дополнительными рисками, которые необходимо учитывать, например:

- Количество приобретенных акций или долей слишком мало или слишком велико.
- Законы и нормативные акты нарушаются из-за неверных данных о покупках и продажах в журналах аудита.

- Акции или доли не покупаются своевременно, то есть кто-то другой успевает купить их раньше нас.

Эти риски могут привести к тому, что стратегия тестирования будет иметь другой приоритет или фокус, как показано на рис. 7.6.

Рис. 7.6. Стратегия тестирования для приложения, связанного с акциями и биржей

Приведенный пример является чрезмерным упрощением гипотетического проекта, но он демонстрирует, что, следуя той же схеме определения характеристик качества, рисков и мероприятий по тестированию, мы можем создать правильную стратегию тестирования для нашего продукта и потребностей команды.

7.2. ПРЕВРАЩЕНИЕ СТРАТЕГИИ ТЕСТИРОВАНИЯ В ПЛАН ТЕСТИРОВАНИЯ

Разработка стратегии определяет путь, по которому мы собираемся идти, но ей не хватает детализации. Она показывает ви́дение нашего тестирования, но если мы действительно хотим достичь поставленных целей, нужен конкретный план. Планирование деталей того, как мы собираемся реализовать нашу стратегию тестирования, очень важно, потому что видение почти всегда вступает в конфликт с реальностью, как только мы начинаем работать. Возможно, мы хотим создать новую систему автоматизации, но понимаем, что не имеем достаточных навыков для ее разработки. Или мы хотим начать проводить собрания по тестированию проекта API, но наша команда разбросана по разным часовым поясам.

Составив план, мы определим ограничения, существующие в нашем контексте, и определим конкретные задачи по успешной реализации стратегии тестирования. В процессе планирования нужно соблюдать следующие принципы:

- *Определите ограничения.* Успех стратегии тестирования будет основан на нашей способности работать в существующем контексте. Вот почему, прежде чем мы начнем составлять план, важно проанализировать пригодность контекста для тестирования. Бессмысленно пытаться работать в отрыве от контекста, например, тестировать дизайн API, когда команда работает удаленно и нет формальных встреч по проекту API.
- *Составьте план и донесите его до коллег.* Планируя задачи тестирования, мы должны рассказать о том, как собираемся их достичь. Информация должна быть сбалансированной и содержать необходимое количество деталей нашего плана.
- *Составьте поэтапный план.* Если пытаться спланировать реализацию всей стратегии тестирования в целом, то такой план будет сложно выполнять и контролировать. Планирование конкретных мероприятий позволяет получить управляемый план, требующий меньших затрат времени на составление и контроль. Если на каком-либо этапе произойдет сбой, мы сможем быстро среагировать нужным образом.
- *Находите время для размышлений и анализа.* Наши планы не всегда будут идти так, как мы хотим. По мере продвижения к цели мы будем узнавать все больше о контексте работы. При этом сможем преодолевать проблемы по мере их возникновения.

Давайте уделим немного времени каждому из этих пунктов и посмотрим, как они помогут превратить видение стратегии тестирования в четкий план, которому будет следовать команда.

7.2.1. Понимание тестируемости контекста

Под тестируемостью мы понимаем все, что влияет на процесс тестирования, — от самого продукта до людей, вовлеченных в проект. Оценка тестируемости контекста позволяет использовать положительные аспекты и уменьшать влияние отрицательных.

Первый шаг оценки тестируемости — понять, какие именно факторы следует учесть. К счастью, у нас есть несколько отличных источников по этому вопросу, например «10П тестируемости» (10 Ps of Testability), которые были созданы

Робом Мини (Rob Meaney) и Эшем Уинтером (Ash Winter). Роб и Эш глубоко изучили тему тестируемости и формализовали различные факторы в модели, представленной на рис. 7.7.

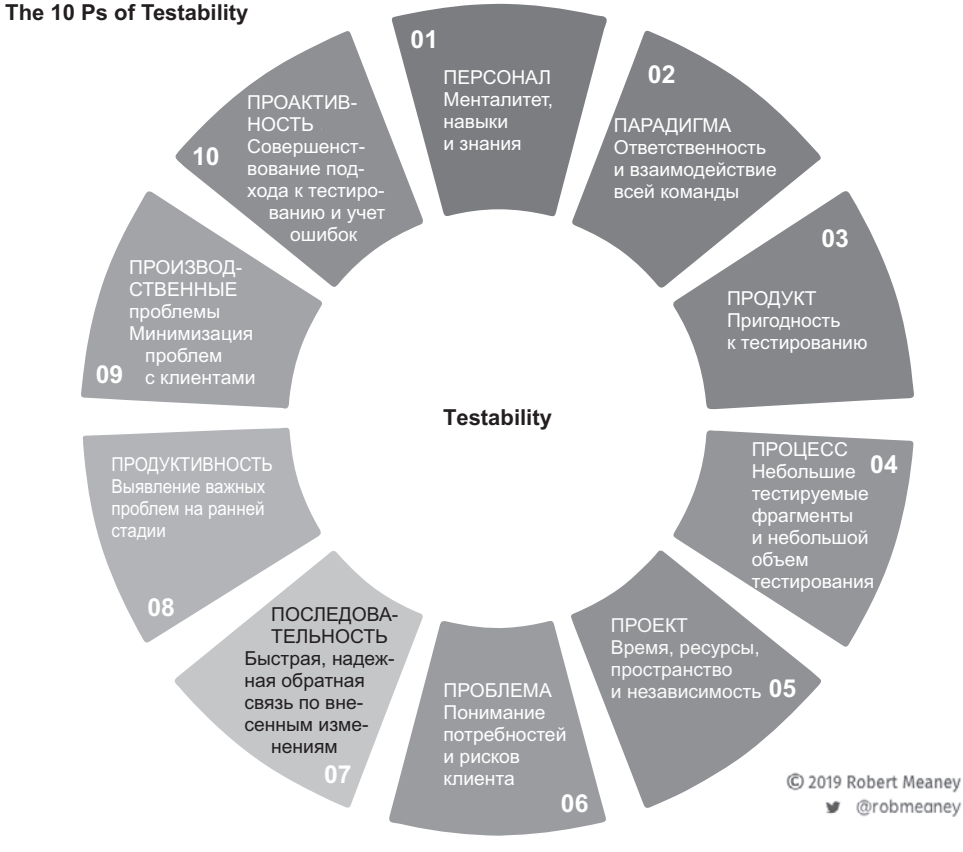

Рис. 7.7. Модель «10П тестируемости», созданная Робом Мини и Эшем Уинтером

Роб и Эш предложили эту модель в книге *«Team Guide to Software Testability»* (https://leanpub.com/softwaretestability), в которой подробно рассматривается тестируемость. Чтобы начать анализ, мы можем использовать их принципы тестируемости:

- *Персонал*. На процесс и результаты тестирования влияют наши навыки, знания и образ мышления. Если мы обладаем ограниченными навыками, то и наши способности к тестированию ограниченны. Поэтому при планирова-

нии работы нам необходимо учитывать, насколько сложно членам команды выполнять конкретные действия по тестированию.

- *Парадигма*. Если пункт «Персонал» фокусируется на способностях и влиянии отдельных людей, то «Парадигма» касается убеждений и установок команды в целом. Например, если команда не очень хорошо понимает, что такое качество и ценность тестирования, это затруднит реализацию стратегии тестирования.
- *Продукт*. Дизайн продуктов и используемые технологии также оказывают огромное влияние на процесс тестирования. Если продукт трудно развернуть, к нему нет доступа или он постоянно дает сбои, то применение инструментов тестирования будет затруднено. При планировании мероприятий по тестированию мы должны убедиться, насколько продукт приспособлен к ним.
- *Процесс*. Как организована работа нашей команды? Доработки выполняются небольшими управляемыми частями или все делается в больших релизах? Если релизы выпускаются редко и содержат большое количество изменений, это может затруднять автоматизацию, приводить к бешеной спешке и затруднять определение всех приоритетных рисков.
- *Проект*. Продолжительность и финансирование проекта также влияют на наш план. Если сроки проекта короткие, у нас может не хватить времени на реализацию конкретных мероприятий по тестированию. Если нам потребуется провести обучение членов команды, будет ли выделен необходимый бюджет и время?
- *Проблема*. Важно хорошо понимать проблемы, которые должно решить программное обеспечение. Также мы должны понимать, как пользователи взаимодействуют с нашей системой. Если знания в любой из этих областей недостаточны, то в тестировании, скорее всего, будут пробелы.
- *Последовательность*. Если мы хотим внедрить определенные процедуры тестирования, например автоматизацию, какое место они займут в общем процессе создания и развертывания продукта? Существует ли вообще единая последовательность операций? Знание процесса создания продукта поможет спланировать, когда и где мы хотим проводить его тестирование.
- *Продуктивность*. Насколько эффективно наша команда решает проблемы? На продуктивность могут влиять как долгосрочные факторы, такие как корпоративная культура, так и краткосрочные, например болезнь сотрудника, объем совещаний в определенные периоды времени и даже психическое состояние команды. Мы не можем учесть всего этого, но знание о соответствующих факторах поможет предвидеть некоторые проблемы и планировать их решение.

- *Производственные проблемы*. Малое число производственных проблем означает, что у нас есть больше времени на тестирование (хотя это может означать и то, что у нас плохая обратная связь). В то же время, когда мы сталкиваемся с серьезными производственными проблемами, это сокращает возможности тестирования. Понимание того, как часто возникают производственные проблемы и как мы их решаем, позволяет определить, сколько времени у нас есть для тестирования.
- *Проактивность*. Насколько мы как команда открыты для регулярного улучшения процессов тестирования? Какие возможности есть для улучшений? Обратная связь с пользователями и рефлексия жизненно важны для оценки того, насколько успешны наши стратегии и планы.

Каждый из пунктов модели 10П позволяет узнать больше о том, что помогает или мешает нашему тестированию. Это важно, поскольку проблемы, влияющие на тестируемость, не всегда очевидны. Например, однажды я работал над проектом, в котором возникли проблемы с созданием тестовой среды. Эту проблему можно было бы отнести к пункту «Последовательность» модели 10П, но анализ показал, что корнем была борьба двух высокопоставленных руководителей за бюджет на следующий год. Они намеренно нарушали работу отделов друг друга, чтобы получить больший бюджет, то есть проблема относилась скорее к категории «Парадигма».

Не все проблемы тестируемости столь экстремальны, но анализ тестируемости контекста помогает выявлять и устранять потенциальные препятствия. Например, если мы хотим внедрить новый инструмент автоматизации, но анализ тестируемости показывает, что команда имеет ограниченный опыт таких внедрений, то мы можем учесть эти знания в планах и попытаться решить проблему путем обучения.

Альтернативные модели тестируемости

«10П тестируемости» Роба и Эша — это, конечно, не единственная модель, помогающая командам исследовать тестируемость. Популярными моделями также являются «Эвристика тестируемости программного обеспечения» (Heuristics of Software Testability) Джеймса Баха (James Bach) (http://mng.bz/K0AX) и «Измерения тестируемости» (Dimensions of Testability) Марии Кедемо (Maria Kedemo) и Бена Келли (Ben Kelly) (http://mng.bz/9V7j). Изучение каждой модели может быть полезным. Хотя они имеют некоторые общие черты, но разное представление тестируемости стимулирует дополнительные вопросы и улучшит понимание проблемы.

Понимание тестируемости помогает не только справиться с проблемами, мешающими тестированию, но и скорректировать приоритеты. Если мы обнаружим, что тестируемость нашего контекста низкая и что тестирование, которое мы хотим провести, без значительных инвестиций не достигнет успеха, то возможно, захотим пересмотреть свои планы. Можно оценить соотношение стоимости работ и потенциальных результатов и скорректировать список рисков в пользу более простых и легких для устранения. Это не означает полный отказ от первоначальных планов, но мы должны подходить к стратегии с пониманием того, что некоторые направления работы могут потребовать высоких инвестиций. Не следует класть все яйца в одну корзину (план), рискуя потерпеть полную неудачу и начать все сначала.

7.2.2. Организация и документирование плана

Традиционно план тестирования состоял из объемных, подробных документов, составители которых пытались охватить все детали. Огромные, трудно поддающиеся расшифровке документы сложно поддерживать актуальными в условиях неизбежных изменений. Поэтому неудивительно, что появилась тенденция, когда команды отказываются от составления планов тестирования, ссылаясь на принцип манифеста agile:

> *Работающий продукт важнее исчерпывающей документации.*

Хотя этот принцип иногда неверно истолковывают как призыв вообще отказаться от документации, на самом деле он пропагандирует рациональный, ценностно-ориентированный подход к ней. Нормально иметь документацию, пока она позволяет отдельным людям и командам создавать работающее программное обеспечение. Этот образ мышления можно и нужно применять к тестовой документации. Можно найти баланс при составлении плана тестирования, не перенасыщая его деталями и не делая слишком расплывчатым. За это выступают Лиза Криспин (Lisa Crispin) и Джанет Грегори (Janet Gregory) в своей книге *«Agile Testing: A Practical Guide for Testers and AgileTeams»* (издательство Addison-Wesley Professional, 2008):

> *Подумайте об облегченном плане тестирования, который охватывает все самое необходимое, но без излишеств.*

Лиза и Джанет предлагают идею создания плана тестирования, в котором рассматривается только необходимая информация и который может быть изложен всего на одной странице. Идея одностраничного плана тестирования была расширена

Клэр Реклесс (Claire Reckless) в статье *«The One-Page Test Plan»* (https://www.ministryoftesting.com/articles/3b0d6d34?s_id=14978711), и там критикуются большие планы тестирования:

> *Предложите очень занятому менеджеру документ на многих страницах, переполненный информацией, на чтение которого может уйти час или больше, и он так и не найдет времени на ознакомление. Представьте короткий документ с описанием запланированного для проекта тестирования, и такой документ с большей вероятностью будет изучен.*

Клэр исследует различные подходы к созданию короткого и простого плана, который передает необходимые детали и ничего более, начиная от простых одностраничных документов Word и заканчивая дашбордами. Независимо от формата, главной целью всегда является создание плана, который четко излагает наши намерения команде. Или, как выразились Лиза и Джанет:

> *Какой бы план тестирования ни был принят в вашей организации, сделайте его своим. Используйте его так, чтобы он приносил пользу вашей команде и удовлетворял потребности клиентов.*

Если мы хотим создать план тестирования, который будет работать на нас и не будет слишком сложным, что мы должны в него включить? Давайте рассмотрим некоторые детали, которые можно добавить на примере стимулирования команды к занятию тестированием проекта API:

- *Введение*. Во введении рассказывается о том, что представляет собой план тестирования и какие области он охватывает. Для нашего примера мы можем указать, что это план для реализации командного подхода к тестированию проекта веб-API.
- *Риски*, которые могут повлиять на наши планы и которые выявляются при оценке тестируемости контекста. В нашем примере можно выделить риски, связанные с отсутствием у команды опыта совместных обсуждений или трудностями подключения к обсуждению членов команды, работающих удаленно.
- *Предположения*. Наряду с рисками, результатами анализа тестируемости могут быть предположения о том, насколько хорошо мы можем тестировать. Добавление их в план позволяет разобраться в ходе тестирования, являются ли они обоснованными. Например, мы можем сделать предположение, что наша команда уже ведет неформальные обсуждения проекта API, чем мы можем воспользоваться.

- *Инструменты*. Какие инструменты мы хотим использовать в рамках нашего плана? Например, в главе 4 мы обсуждали применение Swagger для документирования API, поэтому здесь мы можем указать инструменты Swagger.
- *Ресурсы*. Перечисление дополнительных ресурсов. Здесь может быть отображено время, которое требуется отдельному человеку или команде для реализации плана, а также необходимость обучения и дополнительного финансирования. Для нашего случая можно предусмотреть дополнительный инструктаж или обучение, например, тестированию дизайна API, чтобы помочь внедрить новый подход к тестированию.
- *Сфера деятельности*. Здесь мы можем сообщить, что входит или не входит в область применения нашего плана. Уточнение этих деталей поможет четко передать желаемые результаты, а также выявить области тестирования, которые необходимо рассмотреть в будущем. Например, мы можем указать, что в сферу деятельности входит тестирование веб-API, которые мы разрабатываем, а не API, от которых мы зависим.

Мы можем использовать и другие варианты плана. Анализ тестируемости поможет определить, что необходимо донести до пользователей, чтобы они могли четко понять наши намерения и помочь нам их реализовать.

7.2.3. Выполнение плана и его осмысление

При обсуждении стратегий и планов почти клише — цитировать Роберта Бернса, поскольку одна из его самых известных цитат напоминает нам о важном моменте в планировании:

> *Лучшие планы мышей и людей часто идут вкривь и вкось.*

Повторим сказанное в начале этой главы: мы не можем тестировать всё. То же самое можно сказать и о планировании тестирования: мы не можем планировать все возможные варианты. Анализ тестируемости контекста требует много времени, а значит, у нас не всегда будет возможность заранее выявить все влияющие факторы. Контекст меняется, бюджеты растут или уменьшаются, сроки проекта сдвигаются вперед или назад, члены команды меняются. Все эти события будут влиять на наши планы, поэтому важно учитывать их таким образом, чтобы выявлять проблемы и быстро на них реагировать.

Выполнение

При реализации наших планов мы можем взять за образец современную практику разработки программного обеспечения, разделив работу на небольшие

управляемые задачи. Вместо того чтобы пытаться внедрить новую стратегию тестирования сразу целиком с полным циклом реализации, лучше использовать гибкий подход. Небольшие эксперименты помогут проверить, насколько успешны новые идеи и подходы, а затем можно рассмотреть следующий шаг их дальнейшего развития.

Например, в главе, посвященной тестированию дизайна API, мы узнали, что собрать команду для совместного обсуждения работы API — это возможность проверить идеи и уточнить предположения. Однако идея собрать всю команду на формальное собрание для обсуждения проекта может оказаться сложной с организационной точки зрения. К тому же она может породить враждебность, если встреча пройдет без успеха (знаю это по собственному опыту). Вместо этого, делая небольшие шаги, мы получим время, чтобы дать другим людям привыкнуть к новому подходу и самим вовремя реагировать на изменения или сопротивление. Можно предложить следующие шаги:

1. Проинформируйте команду о намерении начать тестирование дизайна API и опросите ее членов, чтобы выяснить, есть ли интерес к участию.
2. Начните с неформальной встречи с заинтересованными коллегами, сообщив, что это эксперимент, и посмотрите на результаты.
3. Проанализируйте проведенную встречу, внесите необходимые изменения и повторите попытку.
4. Поделитесь успехами с командой, попытайтесь сделать встречи частью регулярного процесса и пригласите других присоединиться.

Каждый шаг может потребовать нескольких итераций и, возможно, не учитывает сложности изменения работы команды. Однако после выполнения каждого шага у нас будет возможность изменить план. Если на первом этапе окажется, что никто не заинтересован в сотрудничестве, то возможно, придется применить другой подход. Главное, чтобы мы учились и адаптировались как можно быстрее, и успех нашим планам обеспечен.

Размышление

Если мы не будем анализировать ход реализации наших планов и их фактическое выполнение, то все усилия окажутся напрасными. Поэтому необходимы механизмы, которые позволяют оценить проделанную работу и при необходимости изменить курс.

Церемония «ретроспектива» предоставляет возможности для осознания, чего мы хотим достичь в ближайшие дни или недели с помощью тестирования, а также

оценки того, как продвигается наша работа. Напомню, что цель тестирования — поддержать усилия команды по улучшению качества продукта. Собрав команду в одном месте, чтобы обсудить ход работы, мы получаем прекрасную возможность провести опрос и узнать, помогают ли коллегам наши планы. Также можно провести встречу в неформальной обстановке, чтобы оценить изменения качества продукта. Улучшилось ли оно? Ощущают ли коллеги, что изменения в стратегии и планах тестирования способствовали улучшению?

Использование таких моделей проведения экспериментов, как *Popcorn Flow* Клаудио Перроне (Claudio Perrone) (https://agilesensei.com/popcornflow/), позволит структурировать процессы внедрения идей, оценки результатов и обучения. Однако такие подходы требуют поддержки со стороны коллег, чтобы они спокойно относились к регулярным экспериментам и обсуждениям.

Наконец, простые беседы могут принести полезные результаты. Иногда информированность о тестировании недостаточна, что приводит к непониманию его сложности и значимости. В ходе беседы можно объяснить задачи и результаты тестирования, а также получить обратную связь, чтобы более эффективно помогать членам команды.

7.2.4. Развитие стратегии

Последовательно выполняя оценку тестируемости, разработку плана, его реализацию и, наконец, анализ результатов, мы поймем, как видение стратегии тестирования преобразуется в реальность. Однако поскольку наша работа со временем меняется, изменяются и влияющие на нее риски. Если мы будем регулярно анализировать значение качества для пользователей и сопутствующие риски, наши приоритеты в выборе видов тестирования будут меняться. Будь то новая стратегия или уже существующая, следование описанной выше схеме помогает определить, на чем сосредоточить нашу энергию в данный момент, а что можно отложить на будущее.

ИТОГИ

- Модели образа и реализации продукта, а также характеристик качества и рисков помогают определиться со стратегией тестирования.
- Риски могут быть уменьшены с помощью тестирования. Чем выше приоритет риска, тем больше оснований для выполнения соответствующих тестов.

- Схема разработки стратегии может быть обобщена следующим образом: определение характеристик качества; анализ рисков, которые влияют на качество; выбор мероприятий, которые помогают снизить риски.
- Стратегия тестирования помогает определить концепцию, но также нужен план или планы, чтобы обеспечить ее выполнение.
- Успех наших планов и стратегии будет зависеть от тестируемости или способности тестировать в существующем контексте.
- Мы можем использовать модель 10П, которая поможет проанализировать различные аспекты контекста, чтобы обнаружить проблемы тестируемости.
- Понимание тестируемости помогает составить лучший план действий и изменить приоритеты, если предстоит решить сложные вопросы.
- План тестирования не обязательно должен быть сложным документом, в котором все подробно описано. Одностраничный план тестирования с необходимыми деталями поможет эффективно донести наши намерения.
- При выполнении планов начинайте с небольших управляемых задач и действуйте поэтапно, это поможет своевременно реагировать на проблемы.
- Церемония «ретроспектива» и экспериментальные модели, такие как Popcorn Flow, способствуют анализу достигнутого.

Часть 3

Расширяем нашу стратегию тестирования

Методы тестирования, о которых мы говорили выше, являются важной частью общей стратегии, но существуют и другие возможности. Например, тестирование производительности и безопасности на заключительной стадии проекта. Как мы узнаем, такого рода мероприятия могут и должны быть включены в стратегию как можно раньше. Они не просто расширят наше понимание работы веб-API и того, как происходит взаимодействие с ним на этапе продакшена. Они также являются прогрессивными способами развития наших возможностей тестирования.

В главе 8 мы познакомимся с такими способами и узнаем, как повысить уровень автоматизации с помощью специальных инструментов и методов. В главе 9 обсудим, как контрактное тестирование помогает автоматизации и как улучшить сотрудничество команд при разработке API. В главе 10 рассмотрим тестирование производительности и то, как начать планировать и реализовывать его значительно раньше, чем это принято. В главе 11 поговорим о том, как включить тестирование безопасности в моделирование, исследовательское тестирование и автоматизацию. В завершение этой части и книги в целом мы узнаем, как тестирование в продакшене и методы проектирования надежности сайта помогают развивать стратегию тестирования и внедрять ее в реальные проекты.

8 Продвинутая автоматизация веб-API

В этой главе

- ✓ Использование автоматизации для управления развертыванием
- ✓ Повышение надежности автоматизации с помощью заглушек (mocks)
- ✓ Добавление автоматических проверок в пайплайн сборки

По мере развития технологии веб-API развивались и инструменты автоматизации. В главе 6 мы узнали, как создавать автоматические проверки API, которые помогают выявлять потенциальные проблемы, но мы можем сделать гораздо больше. В этой главе мы будем опираться на наши знания об автоматизации проверок API и рассмотрим, как мы можем перевести автоматизацию на новый уровень. Для этого в каждом разделе главы будет рассмотрен способ, с помощью которого можно получить больше от автоматизации. Например:

- Управление развертыванием с помощью автоматизированного приемочного тестирования.
- Сокращение количества ложных срабатываний при автоматизации с помощью заглушек.

- Использование тестирования на основе свойств для обнаружения неожиданных проблем.

Допущения и вспомогательная кодовая база

Некоторые действия в этой главе основаны на автоматических проверках, созданных в главе 6. Предполагается, что вы уже изучили главу 6 и создали собственный код, который теперь можно обновить. Если вы этого не сделали, я рекомендую прочитать главу 6, прежде чем продолжать. Кроме того, вы можете получить копию кодовой базы для соответствующего репозитория здесь: http://mng.bz/WMjg.

8.1. РАЗРАБОТКА ЧЕРЕЗ ПРИЕМОЧНОЕ ТЕСТИРОВАНИЕ

Как мы видели в предыдущих главах, различные виды тестирования помогают снизить риски, и автоматизация проверок API не является исключением. Созданные нами ранее автоматизированные проверки API были сосредоточены на рисках, которые мы определили на основе знаний о системе. Например, что, если мы не получим в ответе правильный код состояния либо изменим что-то в системе и это приведет к возвращению неправильного кода состояния? Эти риски относятся к категории «регрессионных», которые сосредоточены на том, что система может измениться нежелательным образом по мере добавления новых функций и модификаций ее кода.

Однако также существуют инструменты, которые помогают снизить риски, связанные с неправильным развертыванием. Они несколько отличаются от регрессионных рисков, поскольку в центре внимания находится не то, как система может «испортиться», а то, что мы (команда) можем неправильно понять особенности продукта, который должен быть развернут. Разработка через тестирование помогает решить эту проблему.

В разработке через приемочное тестирование (ATDD, acceptance test-driven development) и разработке через тестирование (TDD, test-driven development) мы можем использовать одну и ту же модель:

1. Создайте автоматическую проверку, которая заведомо будет провалена вследствие отсутствия соответствующей функциональности.
2. Напишите продакшен-код.
3. Проведите рефакторинг кода для удачного прохождения проверки.

Ожидания от автоматизированной проверки основываются на требованиях заказчика к API. Проверки ATDD имеют тенденцию быть более высокоуровневыми, сфокусированными на бизнес-поведении продукта по сравнению с TDD, где основное внимание уделяется отдельным элементам существующей логики.

В случае ATDD мы начинаем работу с беседы с заказчиком и создаем задачу для автоматизации, например следующую:

- Функция: отчеты о бронировании.
- Сценарий: пользователь запрашивает общий доход по всем заказам:
  - *Given*: учесть возможность нескольких бронирований.
  - *When*: когда поступил запрос отчета об общих доходах.
  - *Then*: затем сообщается общая сумма, рассчитанная на основе всех заказов.

Этот сценарий, написанный с помощью языка Gherkin[1] (Given-When-Then), может быть использован некоторыми средствами автоматизации для создания заведомо провальной автоматической проверки. Она не работает, потому что производственный код, который должна проверять, еще не существует. Но если мы напишем код, чтобы проверка прошла, то ее успех означает завершение работы — мы создали то, что нужно заказчику, и избежали риска расширения границ проекта.

8.1.1. Настройка автоматизированного фреймворка для приемочного тестирования

Для этой работы мы создадим новый проект Maven, на этот раз с расширенным списком зависимостей, чтобы включить новые библиотеки для запуска автоматических приемочных проверок, как показано ниже:

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.7.1</version>
    </dependency>
    <dependency>
        <groupId>io.rest-assured</groupId>
```

[1] Gherkin — язык описания желаемого поведения системы. — *Примеч. пер.*

```
        <artifactId>rest-assured</artifactId>
        <version>4.3.3</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.12.2</version>
    </dependency>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-core</artifactId>
        <version>2.12.2</version>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>6.11.0</version>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>6.11.0</version>
    </dependency>
</dependencies>
```

В дополнение к библиотекам, которые мы использовали в предыдущей главе, мы добавили две библиотеки из Cucumber. Это позволяет установить связь между нашими примерами сценариев и кодом автоматизации. Например, когда шаг «учесть возможность нескольких бронирований» будет выполнен Cucumber, он запустит автоматический код для создания нескольких бронирований. Для начала мы создадим фреймворк, аналогичный тому, который использовали ранее, со следующими пакетами в тестовой папке:

- `requests`
- `payloads`
- `stepdefinitions`

В `com.example` мы создадим новый класс `RunCukesTest`, который содержит следующее:

```
@RunWith(Cucumber.class)
@CucumberOptions(
    features = {"src/test/resources"}
)
public class RunCukesTest {
}
```

Наконец, нужно добавить пример сценария в наш фреймворк, создав новый файл `BokingReports.feature` в папке `resources`, как показано ниже:

```
Feature: Booking reports

  Scenario: User requests total earnings of all bookings
    Given I have multiple bookings
    When I ask for a report on my total earnings
    Then I will receive a total amount based on all my bookings[1]
```

Это означает, что когда мы запускаем `mvn clean test`, то получаем:

```
io.cucumber.junit.UndefinedStepException: The step "I have multiple bookings"
    is undefined. You can implement it using the snippet(s) below:

@Given("I have multiple bookings")
public void i_have_multiple_bookings() {
    // Напишите здесь код, который превращает приведенную выше фразу
    // в конкретные действия
    throw new io.cucumber.java.PendingException();
}
```

Класс `RunCukesTest` связывает JUnit и наши функциональные файлы вместе, позволяя Cucumber запускать файл `BookingReports.feature` в качестве теста. `UndefinedStepException` выводится, потому что когда Cucumber запускает файл функций, он ищет «определения шагов», которые соответствуют шагу в выполняемой проверке. Мы рассмотрим это подробнее ниже, а пока отметим, что это подтверждает готовность к созданию провальной автопроверки.

8.1.2. Создание заведомо провальной автоматической проверки

Теперь, когда у нас все готово, мы можем начать автоматизацию, создав новый класс в пакете `com.example.stepdefs` под названием `BookingReportsStepDefs` со следующим кодом:

---

[1] Функция: отчеты о бронировании

Сценарий: пользователь запрашивает общий доход по всем бронированиям

Дано: у меня есть несколько бронирований

Когда я запрашиваю отчет о моих общих доходах

Тогда я получаю общую сумму, рассчитанную на основе всех бронирований

```
public class BookingReportsStepDefs {


    @Given("I have multiple bookings")
    public void i_have_multiple_bookings() {
        // Напишите здесь код, который превращает приведенную выше фразу
        // в конкретные действия
        throw new io.cucumber.java.PendingException();
    }

    @When("I ask for a report on my total earnings")
    public void i_ask_for_a_report_on_my_total_earnings() {
        // Напишите здесь код, который превращает приведенную выше фразу
        // в конкретные действия
        throw new io.cucumber.java.PendingException();
    }

    @Then("I will receive a total amount based on all my bookings")
    public void i_will_receive_a_total_amount_based_on_all_my_bookings() {
        // Напишите здесь код, который превращает приведенную выше фразу
        // в конкретные действия
        throw new io.cucumber.java.PendingException();
    }

}
```

Обратите внимание, что к каждому методу прикреплена аннотация, где в качестве параметра передается фраза. Первая аннотация

```
@Given("I have multiple bookings") "
```

(«у меня есть несколько бронирований») соответствует первой строке нашего сценария в файле `BookingReports.feature`:

```
Given I have multiple bookings
```

Именно для этого служит библиотека Cucumber: взять сценарии, написанные обычным языком совместно с представителями заказчика, и привязать каждый шаг к определенному блоку кода автоматизации, который мы хотели бы запустить. Именно отсюда происходит термин *«определение шага»* (step definition) — мы определяем, что конкретно будет выполняться на каждом шаге, чтобы проверить, правильно ли мы строим код. В данный момент, однако, если мы запустим этот код, то получим `PendingException`, поэтому нужно ввести наш код автоматизации. Тогда мы получим заведомо провальную автоматическую проверку.

Начнем с создания нескольких бронирований. Прежде всего, давайте обновим определение первого шага, чтобы оно содержало код для создания заказов:

```
@Given("I have multiple bookings")
public void i_have_multiple_bookings() {
    BookingDates dates = new BookingDates(
        LocalDate.of(2021,01, 01),
        LocalDate.of(2021,03, 01)
    );

    Booking payloadOne = new Booking(
        "Mark",
        "Winteringham",
        200,
        true,
        dates,
        "Breakfast"
    );

    Booking payloadTwo = new Booking(
        "Mark",
        "Winteringham",
        200,
        true,
        dates,
        "Breakfast"
    );

    BookingApi.postBooking(payloadOne);
    BookingApi.postBooking(payloadTwo);
}
```

Далее необходимо создать POJO-объекты в `com.example.payloads`. Давайте создадим класс `BookingDates` и добавим в него следующее:

```
public class BookingDates {

    @JsonProperty
    private LocalDate checkin;
    @JsonProperty
    private LocalDate checkout;

    public BookingDates(LocalDate checkin, LocalDate checkout){
        this.checkin = checkin;
        this.checkout = checkout;
    }
}
```

Затем мы создадим класс `Booking`:

```
public class Booking {

    @JsonProperty
    private String firstname;
    @JsonProperty
    private String lastname;
    @JsonProperty
    private int totalprice;
    @JsonProperty
    private boolean depositpaid;
    @JsonProperty
    private BookingDates bookingdates;
    @JsonProperty
    private String additionalneeds;

    public Booking(String firstname, String lastname, int totalprice, boolean
     depositpaid, BookingDates bookingdates, String additionalneeds) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.totalprice = totalprice;
        this.depositpaid = depositpaid;
        this.bookingdates = bookingdates;
        this.additionalneeds = additionalneeds;
    }
}
```

И наконец, добавим код, необходимый для отправки нашего запроса в `com.example.requests`, в класс `BookingApi`:

```
public class BookingApi {

    private static final String apiUrl = "http://localhost:3000/booking/";

    public static Response postBooking(Booking payload) {
        return given()
                .contentType(ContentType.JSON)
                .body(payload)
                .when()
                .post(apiUrl);
    }
}
```

Если мы запустим автоматическую приемочную проверку с помощью команды `mvn clean test`, то увидим, что первый шаг сценария теперь выполняется, но второй шаг возвращает `PendingException`. Поэтому давайте заполним другие определения следующим образом:

```
private Response totalResponse;

@When("I ask for a report on my total earnings")
public void i_ask_for_a_report_on_my_total_earnings() {
    totalResponse = BookingApi.getTotal();
}

@Then("I will receive a total amount based on all my bookings")
public void i_will_receive_a_total_amount_based_on_all_my_bookings() {
    int total = totalResponse.as(Total.class).getTotal();

    assertEquals(total, 400);
}
```

Чтобы заставить этот код работать, нужно создать в `com.example.payloads` новый класс `Total` со следующим кодом:

```
public class Total {

    @JsonProperty
    private int total;

    public int getTotal() {
        return total;
    }
}
```

Также нужен новый метод в классе `BookingApi` для отправки запроса на конечную точку `Total`:

```
public static Response getTotal() {
    return given()
            .get(apiUrl + "report");
}
```

Это дает нам все необходимое для неудачной автоматической проверки. Когда мы запустим `mvn clean test`, то получим в ответ `java.net.ConnectException: Operation timed out error`.

8.1.3. Добиваемся прохождения автоматической проверки

Следующим шагом будет создание продакшен-кода, чтобы добиться прохождения автоматической проверки. В зависимости от нашей роли в проекте мы либо сами отвечаем за создание рабочего кода, либо поручаем это другим членам команды.

Если продакшен-код завершен и проверка пройдена, значит, мы создали именно то, о чем нас попросили. Затем можно либо выполнить рефакторинг продакшен-кода, заботясь при этом, чтобы проверки по-прежнему были успешны, либо перейти к следующему сценарию, требующему развертывания.

Кроме того, как только мы получили результаты автоматической приемочной проверки, можно добавить ее к другим автопроверкам, которые будут выполняться в рамках пайплайна сборки. Это позволяет создать набор автопроверок, которые пригодятся, если изменения в системе нарушат функциональность, требуемую заказчиком.

8.1.4. Остерегайтесь ловушек

Здесь стоит повторить, что подход к проектированию на основе приемочных тестов отличается от других видов автоматизированного тестирования API, потому что основное внимание уделяется *риску получить не то, что нужно*. Цель ATDD — не исчерпывающая проверка всех рисков и всех комбинаций значений, которые может принимать API, и не замена TDD (они работают бок о бок). Она заключается в том, чтобы определить, чего хочет заказчик, и использовать это в качестве руководства к действию.

Чтобы такой подход приносил хорошие результаты, требуется время, зрелая команда и взвешенный подход. Легко попасть в ловушку, пытаясь использовать подход ATDD для создания и проверки всех функций. Прежде чем мы завершим этот раздел, я хочу поделиться несколькими примерами ловушек, в которые попадают команды при использовании ATDD. Надо иметь их в виду.

Пытаться все сделать в одиночку

Ключ к успеху в ATDD на самом деле лежит не в автоматизации, а в обсуждениях перед созданием сценариев. Важно, чтобы команда регулярно встречалась для обсуждения новых функций и создания сценариев, описывающих эти функции.

Цель таких встреч — устранить недопонимания до начала работы, чтобы команда могла выполнить все правильно с первого раза. Если вся команда согласна с тем, что описано в сценариях, мы можем с уверенностью заняться автоматизацией, зная, что когда они пройдут, это означает, что команда сделала именно то, что требовалось заказчику.

Ловушка возникает, когда кто-то начинает писать сценарии в одиночку. Это может быть другой член команды или вы сами. Проблема в том, что если

обсуждений не было, мы рискуем закрепить наше непонимание результата. Тогда, даже если автоматические приемочные проверки прошли, выполненная работа не будет соответствовать ожиданиям заказчика, что приведет к той ужасной необходимости переделки, которую мы все так ненавидим.

Пытаться автоматизировать все

Собрать людей в одном месте — это хорошее начало, но важно сохранять концентрацию. Цель состоит в том, чтобы получить сценарии, описывающие основные бизнес-функции, которые нас просят реализовать. Иногда нужно зафиксировать негативные сценарии выполнения задания заказчика. Однако не следует стараться предусмотреть все варианты, при которых функциональность может работать неправильно.

Помните, что основное внимание уделяется *рискам, связанным с обеспечением правильной работы*, а не написанию автоматических проверок на все случаи жизни. Если список сценариев не будет отражать суть ожиданий заказчика, это породит большой объем кода, который не имеет никакой пользы и требует дополнительного обслуживания.

8.2. МОДЕЛИРОВАНИЕ (MOCKING) ВЕБ-API

Одной из самых больших проблем при работе с автоматизацией является снижение количества ложных срабатываний, которые мы получаем в результате автоматических проверок. Эти так называемые нестабильные тесты возникают из-за целого ряда проблем с продуктами, которые мы автоматизируем. Двумя наиболее распространенными проблемами являются управление состоянием и сложные зависимости между тестируемым веб-API и другими API в рамках нашей платформы.

В качестве примера давайте рассмотрим автопроверку, написанную нами в главе 6, которая создаст бронирование, войдет в систему как администратор, а затем удалит бронирование:

```
@Test
public void deleteBookingReturns202(){

    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 1 , 1 ),
        LocalDate.of( 2021 , 1 , 3 )
    );
```

```
    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

    Response bookingResponse = BookingApi.postBooking(payload);
    BookingResponse createdBookingResponse =
     bookingResponse.as(BookingResponse.class);

    Auth auth = new Auth("admin", "password");

    AuthResponse authResponse = AuthApi.postAuth(auth).as(AuthResponse.class);

    Response deleteResponse = BookingApi.deleteBooking(
            createdBookingResponse.getBookingid(),
            authResponse.getToken());

assertEquals(202, deleteResponse.getStatusCode());
}
```

Проблема может возникнуть из-за того, что хотя проверка ориентирована на `booking` API, она использует `auth` API, чтобы убедиться, что мы можем авторизовать запрос на удаление. Это можно обобщить в модели зависимостей, показанной на рис. 8.1.

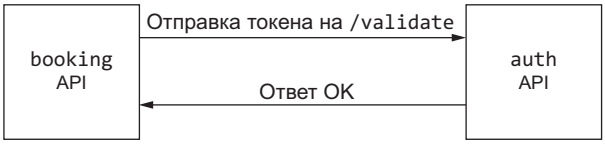

Рис. 8.1. `booking` API отправляет токен `auth` API и получает положительный ответ

Успех процесса удаления бронирования зависит от `auth` API, поэтому проблемы с аутентификацией, будь то неправильно настроенные учетные записи пользователей, ошибки в `auth` API или проблемы со связью между API, приведут к сбою, который не обязательно связан с `booking` API. Это же может привести к постоянному сбою проверки, требующему обслуживания и отладки, что может снизить доверие к платформе.

К счастью, мы можем устранить влияние указанных проблем с помощью библиотек моделирования (мокинга) веб-API, которые помогают изолировать

тестируемый веб-API, а также контролировать поток поступающей в него информации, как показано на рис. 8.2.

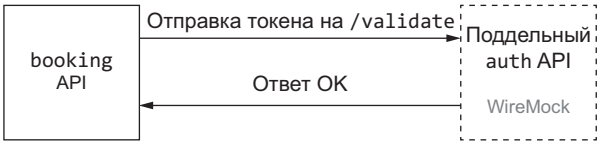

Рис. 8.2. Модель, описывающая `booking` API, отправляющий токен в смоделированный `auth` API

С помощью смоделированного веб-API мы можем определить, какие запросы и в каком формате он должен получать, а также какую информацию отправлять обратно. Чтобы лучше понять, как работает смоделированный веб-API, давайте рассмотрим, как мы можем обновить нашу проверку с помощью инструмента моделирования, или мокинга, — WireMock.

8.2.1. Подготовка к работе

Прежде чем приступить к работе, необходимо сделать несколько подготовительных шагов.

Настройка booking api

Нам потребуется локально запущенная версия `booking` API, на которую мы будем отправлять запросы. Также нужно убедиться, что ничто больше не прослушивает порт 3004, потому что на его использование будет настроен поддельный `auth` API. Для этого загрузите в вашу IDE отдельную копию `booking` API из папки главы 8 в репозитории `api-strategy-book-resources` (http://mng.bz/827K). Теперь вы можете использовать IDE для создания и запуска `booking` API, выполнив метод `main` в классе `BookingApplication`.

Настройка нового кода автоматизации

Сначала мы создадим все необходимое для реализации имитации, начиная с нового примера проверки. Для этого добавьте новую проверку в класс `BookingApiIT`:

```
@Test
public void deleteBookingReturns202WithMocks(){
    BookingDates dates = new BookingDates(
        LocalDate.of( 2021 , 2 , 1 ),
        LocalDate.of( 2021 , 2 , 3 )
    );
```

```
    Booking payload = new Booking(
        1,
        "Mark",
        "Winteringham",
        true,
        dates,
        "Breakfast"
    );

    Response bookingResponse = BookingApi.postBooking(payload);
    BookingResponse createdBookingResponse =
     bookingResponse.as(BookingResponse.class);

    Response deleteResponse = BookingApi.deleteBooking(
            createdBookingResponse.getBookingid(),
            "abc123");

    assertEquals(202, deleteResponse.getStatusCode());
}
```

Обратите внимание, что мы убрали вызовы `auth` API и добавили жестко заданное значение для cookie, которое мы хотим отправить для метода `deleteBooking`. Если бы мы запустили эту проверку прямо сейчас, она, скорее всего, завершилась бы неудачей, поскольку `abc123` не является действительным токеном. Однако мы исправим это, добавив в проверку макет API.

Библиотека мокинга WireMock позволяет программно создавать имитации в нашей кодовой базе. WireMock поставляется с целым рядом полезных функций. В нашем примере мы воспользуемся функциями заглушки, чтобы создать поддельный `auth` API с конечной точкой `/validate`, которая всегда будет отвечать OK, если отправлен правильный поддельный токен. Вы можете узнать больше о других возможностях WireMock на сайте https://wiremock.org/.

Чтобы добавить WireMock, мы создадим зависимость в нашем файле POM.xml:

```
<dependency>
    <groupId>com.github.tomakehurst</groupId>
    <artifactId>wiremock-jre8</artifactId>
    <version>2.30.1</version>
    <scope>test</scope>
</dependency>
```

8.2.2. Построение смоделированной проверки

Теперь, когда API настроены и весь код на месте, давайте обновим нашу проверку, чтобы она использовала WireMock для `auth` API.

Обновление apiurl в booking api

Поскольку мы сейчас работаем с локально развернутой платформой restful-booker, первый шаг — обновить `apiUrl` в классе `BookingAPI` на следующий:

```
private static final String apiUrl = "http://localhost:3000/booking/";
```

Теперь мы будем отправлять запросы на локальный сервер `localhost`, а не на сервер `automationintesting.online`.

Установка wiremock

Теперь настроим WireMock так, чтобы он изображал `auth` API. Для этого требуется добавить следующий код:

```
private static WireMockServer authMock;  <-- Объявление метода WireMockServer

@BeforeAll  <-- Запуск метода перед каждым тестом
public static void setupMock(){
    authMock = new WireMockServer(options().port(3004));  <-- Создание сервера WireMock на порту 3004
    authMock.start();  <-- Запуск имитации
}

@AfterAll  <-- Запуск метода после каждого теста
public static void killMock(){
    authMock.stop();  <-- Остановка имитации
}
```

Итак, у нас есть смоделированный сервер, который принимает запросы через порт 3004. На данный момент у сервера-макета нет конечных точек, поэтому следующим шагом будет создание макета конечной точки.

Создание макета конечной точки

Чтобы создать конечную точку, необходимо добавить следующий код в нашу проверку:

```
authMock.stubFor(post("/auth/validate")
                .withRequestBody(equalToJson("{ \"token\": \"abc123\" }"))
                .willReturn(aResponse().withStatus(200)));
```

Давайте рассмотрим каждый шаг:

1. Мы вызываем `authMock`, который был создан в обработчике `@BeforeAll` до начала проверки.

2. Чтобы создать имитацию конечной точки, вызываем команду `stubFor` и добавляем в нее в качестве параметра детали конфигурации конечной точки.
3. Метод `post()` объявляет конечную точку `POST`, прослушивающую путь `/auth/validate`.
4. `withRequestBody()` задает структуру и тип данных тела запроса. В данном примере мы говорим, что тело запроса должно соответствовать JSON-объекту `{"token": "abc123"}`. Обратите внимание, что значение соответствует `abc123`, которое мы задали для `deleteBooking()`.
5. Метод `willReturn` позволяет указать, какой ответ будет получен, если запрос соответствует ожиданиям в нашей имитации конечной точки. В данном примере требуется только код состояния `200`, который мы устанавливаем с помощью `aResponse().withStatus(200)`.

Создание имитации конечной точки позволяет получить окончательный вариант проверки:

```
@Test
public void deleteBookingReturns202(){
    authMock.stubFor(post("/auth/validate")
            .withRequestBody(equalToJson("{ \"token\": \"abc123\" }"))
            .willReturn(aResponse().withStatus(200)));

    BookingDates dates = new BookingDates(
            LocalDate.of( 2021 , 2 , 1 ),
            LocalDate.of( 2021 , 2 , 3 )
    );

    Booking payload = new Booking(
            1,
            "Mark",
            "Winteringham",
            true,
            dates,
            "Breakfast"
    );


    Response bookingResponse = BookingApi.postBooking(payload);
    BookingResponse createdBookingResponse =
     bookingResponse.as(BookingResponse.class);

    Response deleteResponse = BookingApi.deleteBooking(
            createdBookingResponse.getBookingid(),
            "abc123");

    assertEquals(202, deleteResponse.getStatusCode());
}
```

Проверка отправляет запрос `DELETE` на `/booking/{id}`, который, в свою очередь, возьмет токен из cookie запроса и отправит его в смоделированный API. Если значение маркера совпадает с `abc123`, то имитация вернет правильное сообщение. В противном случае произойдет сбой, что приведет к ошибке проверки.

Теперь мы имеем больше контроля над поведением зависимостей `booking` API. Поэтому если мы столкнемся с какими-либо проблемами, то будем уверены, что они связаны именно с этим API.

8.3. ВЫПОЛНЕНИЕ В СОСТАВЕ ПАЙПЛАЙНА

На протяжении всей этой главы мы рассматривали нетрадиционные способы использования инструментов и библиотек для управления работой, улучшения ее стабильности и обратной связи. Но наступит момент, когда мы захотим запустить автоматизацию как часть пайплайна, который собирает, проверяет и развертывает систему. До сих пор мы выполняли большую часть работы в IDE либо используя развернутые экземпляры веб-API, либо запуская локальные версии веб-API перед выполнением проверок. Это вполне подходит для разработки автоматизации, но что нам делать, когда надо будет запускать проверки в пайплайне?

В отличие от проверки модулей, автоматизация веб-API требует, чтобы он был запущен и работал. Более того, если проверки выполняются как часть другого пайплайна, их нужно запустить в среде, например в блоке непрерывной интеграции (CI), которая отличается от нашей локальной среды. Для этого нам, скорее всего, потребуется внедрить какой-либо инструмент, библиотеку или скрипт. Подход здесь зависит от того, как организована наша кодовая база, какие инструменты мы используем для наших пайплайнов, и многого другого.

Давайте рассмотрим два разных сценария настройки запуска наших автоматических проверок. Один сценарий предполагает, что автоматизация интегрирована с нашей продуктовой кодовой базой, а другой — что автоматизация выделена в отдельный проект.

8.3.1. Автоматизация интегрирована с кодовой базой

Хотя не всегда можно сохранить код автоматизации в проект с рабочей кодовой базой, такое решение, безусловно, рекомендуется. Запуск автопроверок в рамках

процесса сборки проекта (вспомните фазы тестирования и интегрированного тестирования в процессе сборки Maven) не только позволяет быстрее получить обратную связь, но и дает нам более широкий доступ и контроль над настройкой веб-API для запуска проверок.

Программный запуск

Воспользуемся наборами инструментов для тестирования, которые предлагаются в составе библиотек Spring Boot. Поскольку веб-API, с которыми мы работаем, являются частью развитого проекта, предлагающего необходимую основу и инструментарий для создания Java-веб-API, мы можем использовать инструмент `spring-boot-starter-test`, чтобы программно включить наш веб-API.

Давайте рассмотрим расширение примера с имитацией из предыдущего раздела, чтобы нам больше не нужно было самостоятельно «включать» `booking` API. Начнем с добавления необходимой зависимости в файл pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>2.5.4</version>
</dependency>
```

Со `spring-boot-starter-test` на месте мы можем добавить следующие аннотации в верхнюю часть нашего класса `BookingApiIT`:

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT,
    classes = BookingApplication.class)
@ActiveProfiles("dev")
public class BookingApiIT {
```

Давайте рассмотрим каждую аннотацию и посмотрим, как они помогают запустить веб-API:

1. `@ExtendWith(SpringExtension.class)` используется для соединения Junit5 со Spring, чтобы мы могли передавать детали из класса Junit в веб-API, что мы и делаем в следующей строке.
2. Обработчик `@SpringBootTest` служит для настройки запуска `booking` API. Мы используем два параметра: `webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT`, который дает указание `SpringBootTest` загрузить определенный порт из нашего файла `.properties`, а также `classes = BookingApplication.class`, который сообщает `SpringBootTest`, какой класс содержит метод `SpringApplication.run()`, применяемый для запуска веб-API.

3. Строка `@ActiveProfiles("dev")` позволяет задать, с каким файлом `.properties` мы хотим загрузить веб-API. Это полезно, когда у нас есть различные файлы свойств, которые настраивают веб-API на «производственный» или «тестовый» режим (например, увеличение или уменьшение количества записей в журнал в зависимости от того, какое состояние требуется). Передавая параметр `dev`, мы загружаем в API файл `application-dev.properties`.

С этим кодом мы готовы запустить проверку `deleteBookingReturns202()` в нашем классе. Если `booking` API отключен, запустим проверку и увидим подробности запуска API в журнале.

Включение в пайплайн

Поскольку `spring-boot-starter-test` позаботился о настройке веб-API для автоматизации, процесс включения в пайплайн довольно прост. Предполагая, что в файле pom.xml установлен отказоустойчивый плагин (как показано далее в примере), мы запустим команду `mvn clean install` и увидим, как `BookingApiIT` запускается с `booking` веб-API:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.22.2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>integration-test</goal>
                        <goal>verify</goal>
                    </goals>
                    <configuration>
                        <includes>
                            <include>**/*IT</include>
                        </includes>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build
```

Запуск веб-API теперь интегрирован в процесс сборки Maven, и нам остается только настроить используемый инструментарий CI для запуска `mvn clean install`.

8.3.2. Автоматизация в отдельном проекте

Хотя всегда полезно иметь продакшен-код и код автоматизации в одном проекте, не каждый контекст позволяет это реализовать. Препятствием может быть организационная структура, например разделенные команды, которые передают завершенные части работы «через забор» другой команде. Иногда приходится использовать программное обеспечение, которое спроектировано таким образом, что основная кодовая база уже скомпилирована (например, при расширении ранее разработанных платформ). Тем не менее мы должны стремиться интегрировать наши разработки в пайплайн.

Можно настроить веб-API программно, но это требует дополнительной работы. Давайте вернемся к примеру с `booking` API и представим, что на этот раз работаем с версией веб-API, скомпилированной в виде JAR-файла. Тогда нужно будет сделать две вещи:

- включить `booking` веб-API;
- выждать, пока `booking` веб-API будет готов к приему запросов.

С первым шагом можно справиться достаточно легко, воспользовавшись существующим кодом или скриптами, которые используются при развертывании системы. Это может быть что-то простое, например команда запуска JAR-файла в командной строке `java-jar example.jar`, или применение таких инструментов, как Docker, для запуска образа, содержащего наш веб-API (что выходит за рамки данной книги).

Сложнее обеспечить, чтобы наша автоматизация не запускалась до тех пор, пока веб-API не сможет принимать запросы, в противном случае будут ложные срабатывания проверок. Скорее всего, придется либо найти готовый инструмент, либо написать код, который будет блокировать пайплайн на определенное время, пока веб-API не будет готов.

Есть много способов решения этой задачи. Например, можно создать утилиту, которая будет частью пайплайна (как приложение `monitor`, которое я создал на NodeJS для платформы restful-booker http://mng.bz/E0Rq). Или же можно расширить фреймворк автоматизации, добавив проверку того, что веб-API готовы к приему запросов, как показано в обработчике `before`, который можно добавить к `BookingApiIT`:

```
private static void waitForApi(String url, int timeoutLimit) throws
     InterruptedException {
    while(true){
```

```
        if(timeoutLimit == 0){
            fail("Unable to connect to Web API");
        }

        try{
            Response response = given().get(url);

            if(response.statusCode() != 200){
                timeoutLimit--;
                Thread.sleep(1000);
            } else {
                break;
            }
        } catch(Exception exception){
            timeoutLimit--;
            Thread.sleep(1000);
        }
    }
}
```

Метод `waitForAPI` принимает параметры `url` и `timeoutLimit`, которые используются для отправки запроса к выбранной конечной точке. Если запрос не проходит из-за ошибки соединения или ответ не приходит с кодом состояния 200, то метод отсчитывает время `timeoutLimit` и ждет секунду перед повторной отправкой запроса. Метод может завершиться одним из двух способов:

1. `TimeoutLimit` достигает `0`, и возникает сообщение `fail`, указывающее на проблему.
2. Ответ `200` прерывает цикл `while`, а значит, можно начинать автоматическую проверку.

Мы можем вызвать метод в обработчике `@BeforeAll`:

```
waitForApi("http://localhost:3000/booking/actuator/health", 20);
```

Обработчик `@BeforeAll` теперь будет проверять, работает ли `booking` API, каждую секунду в течение 20 секунд. Если к этому времени веб-API не будет запущен, проверка завершится ошибкой с сообщением, что API недоступен.

Стоит повторить, что это всего лишь один из многих способов использования инструментов и/или библиотек, чтобы запустить наши веб-API. Основной вывод: нечто подобное надо предусмотреть в любой системе автоматизации проверок, которая не может сама активировать веб-API.

Такой подход может показаться кому-то неудобным, поскольку возникает больше кода для поддержки и новые потенциальные источники ошибок. Вот

почему иногда стоит сделать шаг назад и посмотреть, можем ли мы внести изменения в наш контекст, которые позволят согласовать производственный код и код автоматизации проверок. В будущем это поможет всем сэкономить время.

ИТОГИ

- Мы можем использовать автоматизированное приемочное тестирование для снижения рисков, предоставляя требуемые функции в нужное время.
- Такие инструменты, как Cucumber, позволяют создавать сценарии Gherkin, которые привязываются к заведомо провальной автоматизации. Затем мы можем создать продакшен-код, чтобы сценарий был выполнен успешно.
- В ситуациях, когда тестируемый веб-API зависит от других веб-API, возможны ошибки, не связанные с тестируемым веб-API.
- Заменив зависимые веб-API такими инструментами, как WireMock, мы можем контролировать, какая информация отправляется в тестируемый веб-API, и уменьшить количество ошибок
- Наши веб-API должны быть готовы к работе с автоматизацией проверок, что можно обеспечить программно.
- Если автоматизированные проверки хранятся в том же проекте, что и производственная кодовая база, мы можем воспользоваться преимуществами инструментов и библиотек для запуска наших API, таких, например, как `spring-boot-starter-test`.
- Если автоматизированные проверки работают отдельно от производственного кода, нам придется использовать инструменты или библиотеки для обеспечения работоспособности веб-API на момент запуска проверок.

9 Тестирование контрактов

В этой главе

- ✓ Что такое тестирование контрактов и как оно помогает в работе
- ✓ Как создать и разместить тест контракта потребителя
- ✓ Как создать и проверить тест контракта поставщика

Представьте себе ситуацию, в которой вы и ваша команда являетесь частью крупной организации и отвечаете за определенное количество веб-API в рамках более широкой платформы API. Ваша команда приложила немало усилий для организации надежного тестирования, и вы собираетесь выпустить новую функцию. Переговоры перед началом разработки были продуктивными, сеансы исследовательского тестирования выявили много полезной информации, а все автоматизированные проверки стали частью пайплайна сборки. Вы развертываете API и обнаруживаете, что платформа сразу падает и возвращает ошибки, — что-то пошло не так.

Начинается бешеная отладка и выясняется, что пока ваша команда была занята работой над новыми функциями, другая команда, ответственная за API, от которого вы зависите, изменила конечные точки, и ваши API больше не могут общаться друг с другом. Хотя проблема может быть решена простым обновлением кода API с учетом последних изменений, остается вопрос: кто отвечает за то, чтобы подобная проблема не повторилась? Ответственность вашей команды — держать всех в курсе любых изменений, или же команда, от которой вы

зависите, должна информировать вас об изменениях? Именно эту проблему призвано решить тестирование контрактов.

9.1. ЧТО ТАКОЕ ТЕСТИРОВАНИЕ КОНТРАКТОВ И КАК ОНО ПОМОГАЕТ В РАБОТЕ

Тестирование контрактов можно рассматривать как программное решение проблемы коммуникации. В идеале мы хотим поощрять команды сообщать друг другу об изменениях и требованиях при создании интеграций между API. Однако реальность может быть иной. Целый ряд факторов может затруднить общение друг с другом: организационные или культурные проблемы; расположение команд в разных офисах или странах; сложность платформ, из-за которой трудно понять, на кого влияют изменения. Тестирование контрактов призвано устранить или, по крайней мере, уменьшить влияние этих факторов благодаря использованию механизмов автоматизации, которые помогают четко определить, какие интеграции существуют между API, и предупреждают об изменениях. Лучший способ объяснить, как работает тестирование контрактов, — это представить его в виде визуальной модели (рис. 9.1).

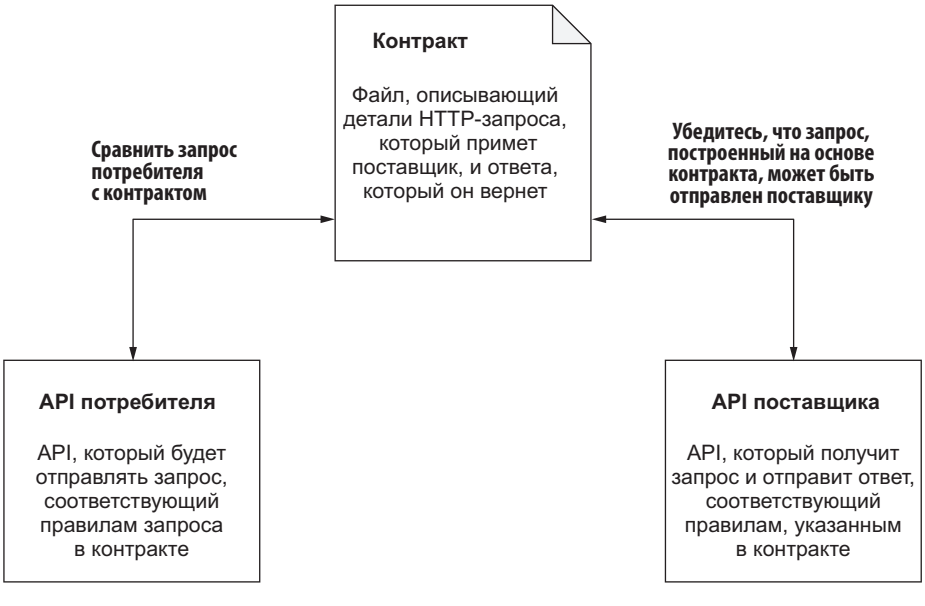

Рис. 9.1. Модель, описывающая взаимосвязи и порядок проведения контрактного тестирования

По сути, тестирование контрактов включает следующие три ключевых компонента:

- *Контракт* — описание правил, установленных для HTTP-запроса и HTTP-ответа.
- *Потребитель* — один или несколько API, которые используют контракт при получении информации из другого веб-API.
- *Поставщик* — API, использующий контракт при «поставке» данных различным веб-API.

Как мы выяснили ранее, чтобы обеспечить интеграцию потребителя и поставщика, контракт должен соблюдаться обеими сторонами. Если потребитель начнет отправлять другие данные или поставщик изменит требования к принимаемым данным, то интеграция не удастся. Поэтому контракт хранится отдельно от API потребителя и поставщика и может быть использован в любой момент для проверки того, что каждый API по-прежнему придерживается контракта.

Допустим, что у нас есть контракт, в котором изложены следующие правила, представленные в виде HTTP-запроса:

```
POST /auth/validate HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "token" : "abc"
}
```

Также контракт предписывает отправлять ответ `200`. Мы можем реализовать следующие инструменты, которые будут проверять соответствие каждого API контракту:

- для API потребителя мы проверяем, что отправляемый HTTP-запрос соответствует правилам контракта: либо путем перехвата такого запроса и его сравнения с контрактом, либо создав имитацию с использованием правил контракта;
- для API поставщика мы создаем и отправляем HTTP-запрос, используя установленные в контракте правила, и проверяем, возвращается ли ожидаемый код состояния.

Если API потребителя или поставщика изменятся и больше не будут соответствовать контракту, то возможны два варианта действий:

- отказаться от изменений, чтобы привести их в соответствие с контрактом;
- обновить контракт.

Если выбран второй вариант и контракт изменен, то тест контракта в другом API завершится неудачей. Это свидетельствует о том, что контракт между API изменился и требуется его обновление.

9.2. НАСТРОЙКА СИСТЕМЫ ТЕСТИРОВАНИЯ КОНТРАКТОВ

Чтобы научиться тестировать контракты, давайте рассмотрим интеграцию между `booking` API и `message` API, которая продемонстрирована на рис. 9.2.

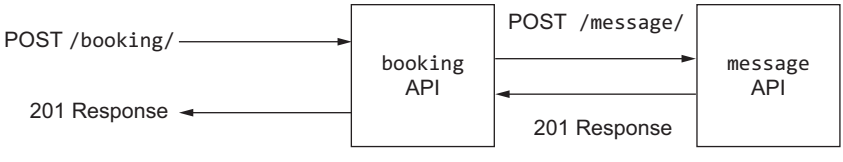

Рис. 9.2. Модель взаимосвязей между `booking` API и `message` API

Как мы видим, `booking` API является потребителем конечной точки `POST /message`, поставщик которой — `message` API. Это означает, что сначала нужно создать тест контракта потребителя для `booking` API, а затем использовать его результаты для создания теста контракта поставщика для `message` API.

Кодовая база

На протяжении этой главы мы будем добавлять тесты контрактов в кодовую базу нашего API. Образцы `booking` API и `message` API имеются в справочном репозитории главы 9 этой книги (http://mng.bz/DD8y). Эти образцы содержат весь производственный и тестовый код, необходимый для создания тестов контрактов, а также готовые версии каждого теста контрактов. Прежде чем мы продолжим, убедитесь, что у вас есть копия кодовой базы, загруженная в IDE. Также предполагается, что вы имеете некоторый опыт автоматизации API или прочитали главу 6 этой книги.

9.2.1. Введение в Pact

Для тестирования контрактов мы будем использовать набор инструментов Pact, созданный компанией Pact Foundation. Pact представляет собой набор инструментов, который обслуживает API как потребителей, так и поставщиков, поддерживает большинство популярных языков и позволяет хранить результирующие контракты, которые совместно используются API. Вы можете узнать больше о Pact из документации на сайте https://docs.pact.io.

В нашей работе мы будем использовать следующие компоненты:

- библиотека потребителя Pact для `booking` API;
- библиотека поставщика Pact для `message` API;
- Pact Broker для хранения наших контрактов и взаимосвязей API.

Взаимодействие этих библиотек можно обобщить, применив их к нашей модели тестирования контрактов (рис. 9.3).

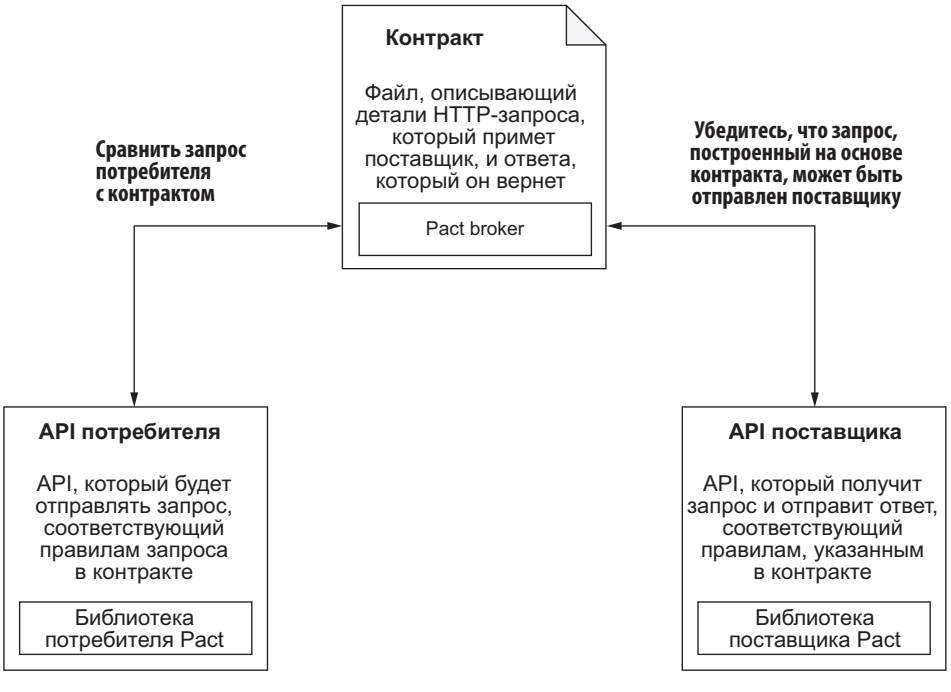

Рис. 9.3. Обобщенная модель тестирования контрактов

Для начала мы создадим тест контракта потребителя для `booking` API, который в случае успеха создаст пакт (pact). Пакт — это JSON-файл, описывающий правила для HTTP-запроса и HTTP-ответа, которые потребители и поставщики используют для интеграции своих API. Он включает такие детали, как используемый URI, тип тела запроса, ожидаемые заголовки и т. д. После создания пакта он может быть опубликован и храниться в Pact Broker. Мы создадим тест для `message` API, который будет извлекать пакт из Pact Broker для проверки соответствия сообщений API правилам. Если установлено соответствие, поставщик сообщит Pact Broker, что пакт проверен.

При первом внедрении Pact в нашу кодовую базу может показаться, что все это очень сложно, но давайте двигаться шаг за шагом, сначала создав тест контракта потребителя.

9.3. СОЗДАНИЕ ТЕСТА КОНТРАКТА ПОТРЕБИТЕЛЯ

Как упоминалось ранее, Pact имеет ряд инструментов для интеграции с разными языками и библиотеками тестирования. Мы использовали JUnit5 в качестве средства запуска тестов, поэтому выберем библиотеку с поддержкой JUnit5, добавив зависимость в файл pom.xml:

```
<dependency>
    <groupId>au.com.dius.pact.consumer</groupId>
    <artifactId>junit5</artifactId>
    <version>4.2.10</version>
</dependency>
```

9.3.1. Добавление Pact к нашему классу

С установленной библиотекой мы можем приступить к разработке теста потребителя, создав в `example.com` новый класс `BookingMessageContractIT`. Теперь добавим следующие аннотации:

```
@ExtendWith(PactConsumerTestExt.class)
@PactTestFor(providerName = "Message API", port = "3006")

@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT,
    classes = BookingApplication.class)

@ActiveProfiles("dev")
public class BookingMessageContractIT {
}
```

Аннотации `SpringExtension`, `SpringBootTest` и `ActiveProfile`, которые мы рассматривали в главе 8, отвечают за программное включение `booking` API перед запуском автоматизации. Другие аннотации являются новыми:

- `@ExtendWith(PactConsumerTestExt.class)` позволяет JUnit подключаться к функциям Pact;
- `@PactTestFor` позволяет определить детали API поставщика, на который опирается `booking` API. В данном случае, поскольку `booking` API полагается на `message` API, мы называем поставщика `Message API` и определяем порт, на котором он будет прослушиваться.

9.3.2. Создание проверки потребителя

В главе 8 мы рассмотрели использование имитаций (моков) в качестве средства создания поддельных API, от которых мы зависим, и смоделировали API со всеми деталями, необходимыми для имитации реальной конечной точки. При создании проверки потребителя мы по тому же шаблону используем Pact для получения мока и определения пакта, или контракта, между двумя API. Для этого мы добавим следующий код:

```
@Pact(consumer="Booking API", provider="Message API")
public RequestResponsePact createPact(PactDslWithProvider builder) {
    MessagePayload message = new MessagePayload(
            "Mark Winteringham",
            "test@example.com",
            "012456789156",
            "You have a new booking!",
            "You have a new booking from Mark Winteringham. They have booked
            a room for the following dates: 2021-01-01 to 2021-01-03");
    return builder
            .uponReceiving("Message")
            .path("/message/")
            .method("POST")
            .body(message.toString())
            .willRespondWith()
            .status(201)
            .toPact();
}
```

Сначала мы объявляем создаваемый нами пакт, используя аннотацию `@Pact` следующим образом:

```
@Pact(consumer="Booking API", provider="Message API")
```

Эта аннотация определяет отношения между потребителем (`booking` API) и поставщиком (`message` API). Если в будущем мы будем создавать новые пакты, затрагивающие отношения с любым из двух API, то используем те же имена, которые задали в качестве параметров в `@Pact`. Тогда при последующей публикации наших пактов они будут сгруппированы под одним и тем же именем API, что облегчит установление отношений между API на наших платформах.

Далее мы создадим `MessagePayload` следующим образом:

```
MessagePayload message = new MessagePayload(
            "Mark Winteringham",
            "test@example.com",
            "012456789156",
            "You have a new booking!",
            "You have a new booking from Mark Winteringham. They have booked
            a room for the following dates: 2021-01-01 to 2021-01-03");
```

Этот POJO-объект помогает определить структуру полезной нагрузки запроса, которую `booking` API будет отправлять в `message` API. Затем мы добавим ее в `PactDslWithProvider` наряду с другими деталями, определяющими наши ожидания относительно настройки конечной точки `message` API. Это можно сделать, например, так:

```
return builder
        .uponReceiving("Message")
        .path("/message/")
        .method("POST")
        .body(message.toString())
        .willRespondWith()
        .status(201)
        .toPact();
```

Как мы видим, `builder` включает:

- *`uponReceiving`* позволяет дать описание на естественном языке того, что отправляется из API потребителя в API поставщика. Оно будет добавлено в документацию Pact, отправляемую нашему Pact Broker. В данном случае мы отправляем простое сообщение, поэтому подойдет имя `Message`;
- *`Path`* — URI для `message` API, к которому, как мы ожидаем, будет направлен запрос;
- *`Method`* — HTTP-метод, который, как мы ожидаем, будет использовать запрос;
- *`Body`* — тело запроса, которое, как мы ожидаем, будет использовано конечной точкой. Поскольку мы создали базовую структуру полезной нагрузки,

мы используем переопределенный метод `toString()`, который преобразует `Message` объекта POJO в следующий JSON:

```
{
    "name" : "Mark Winteringham",
    "email" : "test@example.com",
    "phone" : "012456789156",
    "subject" : "You have a new booking!",
    "description" : "You have a new booking from Mark Winteringham.
    They have booked a room for the following dates: 2021-01-01 to
    2021-01-03"
}
```

Мы завершаем определение, добавляя `willRespondWith()` и объявляя, что поставщик должен возвращать код состояния `201` с помощью `status()`. Затем мы вызываем `toPact()` для завершения создания определения, которое создает пакт между двумя API и настраивает мок. Последний будет использоваться в нашем тесте контракта и выглядит следующим образом:

```
@Test
public void postBookingReturns201(){
    BookingDates dates = new BookingDates(
            LocalDate.of( 2021 , 1 , 1 ),
            LocalDate.of( 2021 , 1 , 3 )
    );

    Booking payload = new Booking(
            1,
            "Mark",
            "Winteringham",
            true,
            dates,
            "Breakfast",
            "test@example.com",
            "012456789156"
    );

    Response response = BookingApi.postBooking(payload);

    assertEquals(201, response.getStatusCode());
}
```

Мы используем подход, рассмотренный в главе 6 при написании простой автоматической проверки, которая создает бронирование и утверждает, что от `booking` API получен положительный ответ. Но «под капотом», как мы видели ранее, `booking` API отправляется сообщение в смоделированную версию `message` API, созданную с помощью Pact. После запуска проверки, если отправленное на мок сообщение соответствует всем критериям, заданным в методе `createPact()`,

booking API получает обратно сообщение 201, что позволяет ему успешно завершить свой процесс и пройти проверку. Но что более важно, в папке target/pacts создается JSON-файл, который документирует пакт, или контракт, между двумя API, как показано ниже:

```
{
  "consumer": {
    "name": "Booking API"
  },
  "interactions": [
    {
      "description": "Message",
      "request": {
        "body": {
          "description": "You have a new booking from Mark Winteringham. They
    have booked a room for the following dates: 2021-01-01 to 2021-01-03",
          "email": "test@example.com",
          "name": "Mark Winteringham",
          "phone": "012456789156",
          "subject": "You have a new booking!"
        },
        "method": "POST",
        "path": "/message/"
      },
      "response": {
        "status": 201
      }
    }
  ],
  "metadata": {
    "pact-jvm": {
      "version": "4.2.10"
    },
    "pactSpecification": {
      "version": "3.0.0"
    }
  },
  "provider": {
    "name": "Message API"
  }
}
```

9.3.3. Настройка и размещение информации в Pact Broker

Мы достигли момента создания пакта, который документирует контракт между booking и message API. Следующим шагом может быть обращение к message API и создание теста контракта для него. Однако предварительно нужно подумать о том, как мы будем хранить файлы Pact, созданные на основе теста контракта потребителя, и обмениваться ими.

При создании теста контракта для API поставщика в Pact мы используем файлы JSON, созданные в тесте контракта потребителя, чтобы задать параметры, по которым тестируется API поставщика (мы рассмотрим это более подробно в ближайшее время, пока же просто нужно понять, что тесты контрактов поставщиков требуют файлов Pact). Это означает, что файлы JSON должны храниться в месте, к которому имеют доступ тесты контрактов поставщиков. Можно использовать простое копирование файлов из `target/pacts` в проекте API потребителя в проект API поставщика. Все усложняется, если команды не имеют доступа к проектам друг друга. Тогда потребуется дополнительная настройка того, какой проект куда копируется. К счастью, команда Pact Foundation подумала об этом и создала Pact Broker.

Pact Broker — это, по сути, API, который хранит файлы JSON (обычно файлы Pact), созданные тестами контрактов. Он предоставляет командам возможность публиковать, просматривать и проверять контракты, что полезно по следующим причинам:

- *Централизация*. Обеспечивается достоверность контрактов.
- *Контроль версий*. Pact Broker предлагает возможность контроля версий контрактов. Это помогает отслеживать историю контрактов и дает потребителям и поставщикам возможность выбирать, какую версию использовать. Также есть возможность проверить изменения, то есть мы можем отслеживать, какие изменения были реализованы, а какие нет.
- *Понимание*. Все контракты хранятся в одном месте, причем Pact Broker предлагает модель взаимоотношений всех наших API, помогая разобраться в зависимостях между ними.

Pact Broker можно установить и запускать в своем проекте, о чем можно узнать из GitHub-репозитория (https://github.com/pact-foundation/pact_broker). Однако для нашего проекта мы будем использовать облачную реализацию Pact Broker от Pact — Pactflow, расположенную по адресу https://pactflow.io. Pactflow позволяет создать бесплатный проект и передать наши файлы Pact в облако.

Первым шагом в настройке Pact Broker является создание аккаунта на сайте https://pactflow.io/, который позволит бесплатно опубликовать до пяти контрактов. После этого у нас появится проект с собственным поддоменом, например, https://restful-booker-platform.pactflow.io. URL необходим, чтобы сообщить нашим проектам, куда отправлять пакты вместе с API-токеном для авторизации действий. Для обновления проекта вам понадобится:

- URL-адрес вашего проекта Pactflow;
- ваш токен API Pactflow, который можно найти в разделе Settings > API Tokens.

С этими данными необходимо обновить проект `booking` API, чтобы он мог публиковать пакты в Pactflow. Добавим плагин в Maven:

```
<plugin>
    <groupId>au.com.dius.pact.provider</groupId>
    <artifactId>maven</artifactId>
    <version>4.1.11</version>
    <configuration>
      <pactBrokerUrl>https://restful-booker-platform.pactflow.io</pactBrokerUrl>
      <pactBrokerToken>TOKEN123</pactBrokerToken> <!-- Replace TOKEN with the
actual token -->
      <pactBrokerAuthenticationScheme>Bearer</pactBrokerAuthenticationScheme>
    </configuration>
</plugin>
```

Как видим, мы добавляем URL нашего проекта Pactflow в элемент `<pactBrokerUrl>`, а токен — в элемент `<pactBrokerToken>`. Когда плагин настроен и наш файл Pact JSON находится в каталоге `target/pacts`, выполним следующую команду в терминале, чтобы опубликовать наш Pact в системе Pactflow:

```
mvn pact:publish
```

После этого должно появиться подтверждение, что публикация прошла успешно. Удостовериться в успешности публикации можно также в Pactflow — в разделе Overview появится неверифицированная интеграция, подобная показанной на рис. 9.4.

Рис. 9.4. Опубликованный неверифицированный контракт в Pactflow

9.4. СОЗДАНИЕ ТЕСТА КОНТРАКТА ПОСТАВЩИКА

Когда тест контракта потребителя пройден и Pact опубликован в Pactflow, мы можем переключить внимание на поставщика. Выполним следующие шаги.

1. Настроим наш проект для подключения к Pactflow.
2. Подключим связанные контракты к API поставщика.
3. Создадим и отправим HTTP-запросы на основе каждого контракта для проверки того, что запрос принят и API отвечает должным образом.

Чтобы лучше разобраться, давайте настроим `message` API с помощью теста контракта поставщика.

9.4.1. Реализация теста контракта поставщика

Как обычно, первым делом мы добавляем в файл pom.xml зависимость Pact для `message` API, как показано ниже:

```
<dependency>
    <groupId>au.com.dius.pact.provider</groupId>
    <artifactId>junit5</artifactId>
    <version>4.2.10</version>
</dependency>
```

Обратите внимание, что `groupId` изменился с `au.com.dius.pact.consumer` на `au.com.dius.pact.provider`.

Установив библиотеку Pact, мы создаем тестовый класс `MessageBookingVerifyIT` в пакете `com.example` и добавляем в класс следующие аннотации:

```
@Provider("Message API")
@PactBroker(url = "https://restful-booker-platform.pactflow.io",
    authentication = @PactBrokerAuth(token = "TOKEN"))

@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT,
    classes = MessageApplication.class)
@ActiveProfiles("dev")
public class MessageBookingVerifyIT {
}
```

Как и в других тестовых классах, которые мы настроили, здесь добавлены аннотации Spring, чтобы включить API перед тем, как начать проверку контрактов. Но также добавлены две новые аннотации:

- `@Provider` настраивает имя API поставщика, с которым мы запускаем тесты контрактов. Обратите внимание, что мы используем в этой аннотации то же имя `Message API`, что и в `@Pact` в тесте контракта потребителя;
- `@PactBroker` настраивает, к какому Pact Broker мы хотим подключиться, включая его URL (в нашем случае — в Pactflow) и API-токен, который предоставляется с помощью аннотации `@PactBrokerAuth`.

Добавив аннотации, мы можем настроить тест контракта, сначала включив обработчик `@BeforeEach`, который добавит в него некоторые детали конфигурации, как показано ниже:

```
@BeforeEach
void before(PactVerificationContext context) {
    System.setProperty("pact.verifier.publishResults", "true");

    context.setTarget(new HttpTestTarget("localhost", 3006, "/"));
}
```

Рассмотрим подробнее, что делает этот обработчик:

- во-первых, мы устанавливаем системное свойство, которое позволяет Pact публиковать результаты проверки нашего контракта обратно в Pactflow. Обычно `pact.verifier.publishResults` имеет значение `false`, поэтому если не добавить этот параметр, то контракты в Pactflow не будут проверены;
- во-вторых, мы передаем параметр `PactVerificationContext` в обработчик, чтобы его можно было обновить с помощью информации о том, где настроен API поставщика. Последнее нужно для запуска наших тестов контрактов.

Наконец, мы завершаем тест контракта, добавляя следующее:

```
@TestTemplate
@ExtendWith(PactVerificationInvocationContextProvider.class)
void pactVerificationTestTemplate(PactVerificationContext context) {
    context.verifyInteraction();
}
```

В отличие от теста контракта потребителя, где мы создавали HTTP-запрос и утверждали HTTP-ответ, здесь мы используем встроенную в JUnit5 функцию `@TestTemplate` и расширяем ее с помощью `PactVerificationInvocationContext Provider`, чтобы позволить Pact динамически создавать JUnit-тест для каждого контракта в Pact Broker, который обозначен как `Message API`.

Когда мы запускаем метод `pactVerficationTestTemplate()`, он загружает все контракты, которые помечены как `Message API` поставщика (это определяется

аннотацией `@Provider`). Далее он создает HTTP-запросы на основе правил в контракте, и если ожидаемый ответ возвращается, контракт считается проверенным. Детали проверки передаются в Pactflow. Результат успешного запуска показан на рис. 9.5.

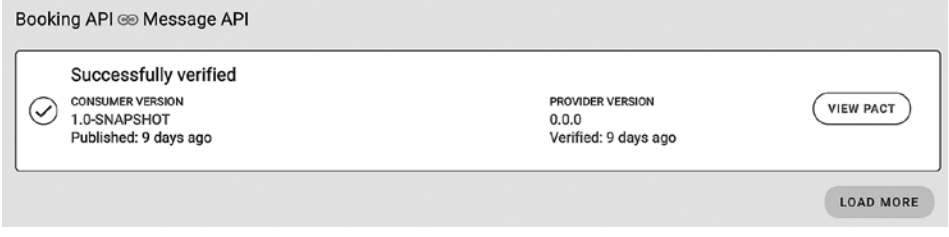

Рис. 9.5. Опубликованный в Pactflow проверенный (верифицированный) контракт

9.4.2. Тестирование изменений

Когда мы установили контракт между `booking` API и `message` API, давайте смоделируем ситуацию, в которой потребитель — `booking` API — изменяет контракт. Для этого вернемся к нашему тесту контракта потребителя в проекте `booking` API и обновим код состояния, возвращаемый моком, который мы создали в методе `createPact`:

```
return builder
            .uponReceiving("Message")
            .path("/message/")
            .method("POST")
            .body(message.toString())
            .willRespondWith()
            .status(200)
            .toPact();
```

Мы изменили код статуса с `201` на `200`. Вроде бы незначительная корректировка, но ее достаточно, чтобы проиллюстрировать процессы при изменениях в контрактах, а также их влияние на интеграцию API. Теперь, запустив тест контракта `postBookingReturns201()`, мы увидим, что JSON-файл в `target/pacts` отражает наше изменение следующим образом:

```
"response": {
  "status": 200
}
```

Это изменение в контракте необходимо опубликовать в Pactflow, снова запустив команду `mvn pact:publish`. Мы можем удостовериться, что обновленный контракт опубликован, поскольку интеграция помечена как Unverified.

Изменив контракт, мы можем вернуться к проекту `message` API и снова запустить `pactVerificationTestTemplate()`. Тест зарегистрирует сбой:

```
Pending Failures:

1) Verifying a pact between Booking API and Message API - Message: has status
     code 200

    1.1) status: expected status of 200 but was 201
```

Как мы обсуждали ранее, это станет стимулом поговорить с командой разработчиков API потребителя, чтобы лучше понять их требования и определить, следует ли обновить код статуса.

9.5. ТЕСТИРОВАНИЕ КОНТРАКТОВ КАК ЧАСТЬ СТРАТЕГИИ ТЕСТИРОВАНИЯ

Особенность тестирования контрактов по сравнению с другими автоматизированными подходами заключается в том, что его можно использовать как для проверки системы, подобно другим рассмотренным подходам, так и для устранения недопонимания между командами. Наша модель стратегии тестирования показывает, что контрактное тестирование помогает снизить риски на стороне представлений в проектах (область воображения), а также подтвердить ожидания (рис. 9.6).

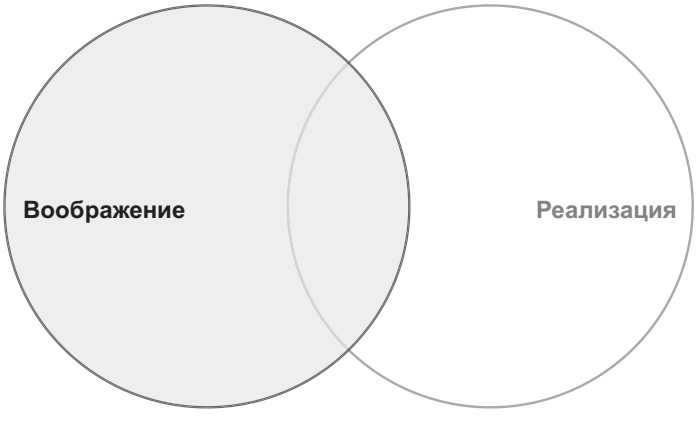

Рис. 9.6. Модель, показывающая, как тестирование контрактов помогает тестированию в области воображения

Неудачный тест контракта на стороне потребителя или поставщика — прекрасная возможность либо улучшить понимание ожидаемой совместной работы API, либо наладить взаимодействие между владельцами API. Иногда недопонимание можно устранить простым обновлением кода, чтобы он соответствовал контракту. Но в других случаях необходимо более широкое обсуждение проблемы между командами, которое, в свою очередь, может выявить новые риски.

ИТОГИ

- Тестирование контрактов — это программный способ обеспечения того, чтобы команды сообщали друг другу об изменениях в API.
- Тестирование контракта проверяет соблюдение контракта как потребителем, так и поставщиком.
- Популярным инструментом для создания тестов контрактов является Pact — набор инструментов, позволяющий запускать тесты для API потребителей и поставщиков.
- Пакты между API могут централизованно храниться в Pact Broker, который обеспечивает контроль их версий.
- Когда тест контракта потребителя пройден, мы публикуем полученный JSON в нашем Pact Broker для проверки поставщиком.
- Тест контрактов поставщика можно настроить на загрузку контрактов из Pact Broker, чтобы проверять соответствие им API поставщика.
- При изменении контракта на стороне потребителя или поставщика проверенный контракт нарушается, а команды информируются о необходимости исправления контракта.

10 Тестирование производительности

В этой главе

- ✓ Как управлять ожиданиями при тестировании производительности
- ✓ Как планировать и проводить тестирование производительности
- ✓ Как выполнить тест производительности и проанализировать результаты

Тестирование производительности демонстрирует, почему мы должны руководствоваться характеристиками качества в стратегии тестирования. Обычно производительность рассматривается как «нефункциональное требование», но как однажды сказал мой коллега Ричард Брэдшоу (Richard Bradshaw): «Если все функции приложения работают, но на ожидание ответа уходит более минуты, то такое приложение кажется мне не очень функциональным».

Проблема различия «функциональных» и «нефункциональных» требований заключается в том, что иногда между ними подразумевается иерархия, определяющая, на каком виде тестирования следует сосредоточиться в первую очередь. Именно поэтому иногда тестирование производительности проводится в спешке, в конце проекта, с ограниченным временем и ресурсами для планирования, реализации и анализа того, как наше приложение работает в конкретном контексте.

Однако если рассматривать производительность как равноценное качество по отношению к другим, более «традиционным» характеристикам, таким как полнота, стабильность и сопровождаемость, то приходит понимание, что производительность в некоторых контекстах является характеристикой качества высокого уровня и она должна иметь приоритет в стратегии тестирования.

В этой главе мы не только узнаем, что такое тестирование производительности и как его проводить, но и разработаем процесс, который позволит последовательно управлять тестированием производительности, чтобы оно стало неотъемлемой частью общей стратегии.

10.1. ПЛАНИРОВАНИЕ ТЕСТА ПРОИЗВОДИТЕЛЬНОСТИ

Самая распространенная ловушка, в которую можно попасть при тестировании производительности, — это отсутствие планирования. Тестирование производительности дает большое количество данных для анализа и интерпретации. Если нет четкого представления о том, что мы хотим узнать, как определить, является ли производительность нашего продукта высокой или низкой? Это, по моему опыту, приводит к ситуациям, когда на основе результатов тестирования производительности делаются предположения без реальных оснований. Следствием становятся необоснованные исправления и многочисленные тесты, которые показывают незначительные улучшения или результаты, противоречащие предыдущим.

Поскольку тестирование производительности требует затрат времени и ресурсов, важно составить план, который учитывает следующие вопросы:

- Какой тип теста производительности мы хотим провести?
- Какие показатели мы хотим измерить при тестировании производительности?
- Каких значений показателей мы хотим добиться от нашего продукта?

10.1.1. Типы тестов производительности

Для начала давайте определим, что мы понимаем под тестом производительности, поскольку этот термин используется в разных контекстах. Тип теста производительности зависит от того, что мы хотим узнать о продукте. Наиболее распространенные типы тестов:

- нагрузочные тесты;
- стресс-тесты;
- тестирование стабильности;
- тестирование базовой версии.

Нагрузочные тесты

Нагрузочный тест помогает понять, как может вести себя наше приложение при предполагаемом количестве запросов. В ходе нагрузочного теста приложение обрабатывает число запросов, которое было спрогнозировано или основано на производственной статистике. Нагрузочный тест помогает оценить, соответствует ли приложение целевым показателям по уровням доступности, параллелизма, пропускной способности или времени отклика (о котором мы скоро поговорим).

Поскольку цель состоит в том, чтобы понять, как работает приложение в условиях, максимально приближенных к реальным, нагрузочный тест обычно включает задержки и паузы, которые возникают при взаимодействии с сайтом, например, когда пользователи заполняют формы или ожидается ответ от других приложений.

Стресс-тестирование

Если нагрузочный тест показывает, как продукт справится с ожидаемой нагрузкой, то стресс-тест должен определить пределы возможностей. Обычно при стресс-тесте постепенно увеличивается количество обращений к приложению, пока объем нагрузки не приведет к появлению ошибок или неприемлемо высокому времени отклика.

Эта информация помогает определить пропускную способность приложения и оценить, является ли она приемлемой. Мы также можем оценить емкость приложения, например то, в какой момент производительность снижается и когда приложение становится недоступным. Этот метод полезен и для понимания того, как приложение справится с DDOS-атаками.

Тестирование стабильности

Не все проблемы производительности связаны с объемом нагрузки, которую испытывает приложение. Они могут появляться со временем из-за утечек памяти или заполнения серверов слишком большим количеством данных. Тест на стабильность заключается в создании ограниченного объема нагрузки в течение длительного времени. В отличие от нагрузочных тестов или стресс-тестов, которые могут занимать один-два часа, тестирование стабильности проводится

в течение многих часов, чтобы попытаться обнаружить проблемы, которые могут проявиться в реальной ситуации. Цель — выявить скачки значений времени отклика или загрузки оборудования (процессора, оперативной памяти, дискового ввода/вывода и т. д.).

Тестирование базовой версии

Тестирование базовой версии (baseline testing) обычно проводится в сочетании с другими типами тестов, такими как нагрузочные и стрессовые. В отличие от других тестов производительности, тестирование базовой версии выполняется с одним виртуальным пользователем определенное количество раз. Полученную информацию можно сравнить с результатами нагрузочных или стресс-тестов для определения степени снижения производительности при увеличении нагрузки на приложение. Например, тестирование базовой версии с одним виртуальным пользователем может показать, что использование процессора составляет 30 %, но когда проводится нагрузочный тест со 100 виртуальными пользователями, использование процессора достигает пика в 35 %. Использование 30 % может считаться не идеальным значением, но сравнение результатов показывает, что приложение способно выдерживать нагрузку.

Таким образом, в ходе тестов мы можем получить самую разную информацию, от значения предельной нагрузки, которую может выдержать приложение, до того, как оно справляется с нагрузкой в течение длительного времени. Нередко каждый тип тестов является частью более широкой стратегии тестирования производительности, потому что подходы отличаются не тем, *что* тестируется, а *количеством нагрузки*, которая применяется. Как мы узнаем далее, можно повторно использовать сценарии тестирования производительности, изменяя «профиль нагрузки», который характеризуется количеством виртуальных пользователей и продолжительностью теста.

10.1.2. Виды показателей при проведении тестов производительности

Помимо понимания различных подходов к тестированию производительности, мы также должны знать об измеряемых характеристиках. Последние зависят от того, о чем мы хотим узнать. Наиболее часто определяют следующие характеристики:

- *Доступность*. Когда приложение находится под нагрузкой, мы хотим убедиться, что оно работает и его сервисы доступны. Увеличивая нагрузку, мы можем проверить, работает ли приложение и доступно ли оно. Например, мы можем отследить, появляются ли коды состояния 400 и 500.

- *Время отклика*. Доступность важна, но также важна и скорость, с которой приложение реагирует под нагрузкой. Чтобы понять, как приложение реагирует, мы можем измерить время между моментами отправки запроса и получения ответа. Чем выше среднее время отклика или значения в пределах определенного процентного соотношения (например, наибольшие 10 %), тем медленнее работает система.
- *Пропускная способность* позволяет оценить скорость, с которой происходят события, связанные с приложением, или количество данных, успешно обработанных за единицу времени. Например, мы можем измерить количество запросов, обработанных в течение секунды, и проанализировать, как меняется это значение по мере роста нагрузки — увеличивается или постепенно выравнивается в определенной точке.
- *Использование ресурсов*. По мере того как приложение получает все большую нагрузку, оно будет использовать больше ресурсов процессора, памяти и сети. Мы можем определить процент использования ресурсов определенного типа. Например, можно проследить, какую часть пропускной способности сети потребляет трафик приложения или какой объем памяти используется на сервере, когда активна тысяча пользователей. Эта информация может быть полезна для выявления проблем в сочетании с показателями доступности, времени отклика и пропускной способности.

Это лишь часть из наиболее распространенных характеристик, которые могут нас волновать. Многие инструменты тестирования производительности позволяют отслеживать перечисленные показатели и многое другое. Заманчиво отслеживать все сразу. Проблема, однако, заключается в том, что объем информации, которую мы получаем по каждой из этих характеристик, может быть чрезмерно большим. Поэтому важно установить четкие цели тестирования производительности — определить, что мы хотим отслеживать и что будем игнорировать (возможно, временно).

10.1.3. Определение целей тестирования производительности и ключевых показателей эффективности (KPI)

При определении цели тестирования производительности мы должны помнить как о типе тестирования, так и о том, что мы хотим измерить. Рассмотрим примеры некоторых целей тестирования производительности:

- Доступность должна быть выше 99 % при одновременном подключении 5000 или более виртуальных пользователей.

- Среднее время отклика не должно превышать 1000 миллисекунд при одновременном подключении 200 виртуальных пользователей в течение 24 часов.
- Загрузка сети должна быть менее 50 % при одновременном подключении 2000 виртуальных пользователей.

Как мы видим, каждая цель подразумевает тип тестирования, которое нужно провести. Первый пример предполагает стресс-тест, в то время как второй требует теста на стабильность. В каждой цели также четко указаны количественные значения, которые помогут оценить результаты теста и определить, что нужно делать дальше.

Какова должна быть целевая производительность платформы restful-booker? Как расставить приоритеты? И снова вернемся к характеристикам качества, которые мы определили в рамках нашей стратегии. Наши характеристики были такими:

- интуитивность;
- полнота;
- стабильность;
- конфиденциальность;
- доступность.

Чтобы поставить цель, мы должны оценить, как риски производительности могут повлиять на *стабильность* и *доступность* нашего продукта. Если платформа restful-booker начнет рассылать ошибки при нагрузке, эти две характеристики не будут обеспечены. Поэтому наше измерение логично связать с доступностью (намек на то, что одно и то же слово используется и для характеристики качества, и для измеряемых показателей). Мы должны определить, какие количественные показатели требуются. В примере с платформой restful-booker после анализа можно получить следующее:

- Мы ожидаем, что обычное число пользователей — около 40, поскольку платформа restful-booker ориентирована на малые B&B-компании с небольшой клиентской базой.
- Наши клиенты хотят, чтобы платформа restful-booker была доступна не менее 95 % времени.

Эти сведения могут быть получены из бесед с пользователями или анализа инструментов, фиксирующих их поведение. На основе этой информации мы

можем сформулировать цель тестирования производительности следующим образом:

> Доступность должна быть выше 95 %, когда к системе подключено 40 виртуальных пользователей.

Эта цель указывает, какие шаги мы должны предпринять при создании теста производительности. Мы можем выполнить нагрузочный тест для 40 виртуальных пользователей или провести стресс-тест для большего их числа, чтобы оценить истинные возможности нашей системы. Но прежде чем создавать тест, нам нужно понять, какие результаты укажут, удалось ли нам достичь своей цели. Вот почему, помимо фиксации цели, также необходимо определить ключевые показатели эффективности.

Хотя данные измерений, которые поступают от инструмента тестирования производительности, дают информацию о поведении системы под нагрузкой, они не всегда позволяют понять причины того или иного поведения. Чтобы лучше разобраться в причинах, важно собирать метрики KPI. Мы можем использовать метрики KPI в сочетании с данными инструмента тестирования производительности, чтобы определить, соответствует ли наше приложение целевым показателям.

Основные группы показателей KPI:

- *Низкоуровневые KPI* сосредоточены на общих ресурсах, которые использует все программное обеспечение. К ним относится использование процессора, оперативной памяти, физического дискового пространства, показатели ввода/вывода и сетевого интерфейса, такие как полоса пропускания и пропускная способность.
- *Серверные KPI* включают показатели, которые могут быть собраны с серверов, обеспечивающих работу веб-API, с помощью таких инструментов, как IIS, Spring Boot, Tomcat или Ruby on Rails.
- *KPI базы данных*. Если для приложения актуален уровень персистентности, то можно отслеживать показатели баз данных, таких как MySQL, Oracle, SQL Server или NoSQL/XML. Это поможет узнать, сколько данных хранится в определенный момент времени, какие ресурсы использует база данных, подробную информацию о блокировках, которые установлены на определенные записи в определенные моменты времени.

Отслеживаемые KPI зависят от используемых инструментов и библиотек. Поэтому необходимо провести некоторое исследование, чтобы определить, что мы хотим и можем отслеживать. Например, платформа restful-booker может быть развернута с помощью Docker, следовательно, есть возможность получить подробную информацию

об использовании процессора, памяти и сетевого ввода/вывода. Зная, что эти KPI доступны, мы можем учесть их в плане тестирования производительности.

Связывая вместе цель тестирования производительности и то, какие KPI мы хотим отслеживать, мы получаем необходимые для работы детали. Теперь мы знаем, что хотим измерить при тестировании производительности, а также разобрались с инструментами, необходимыми для сбора и хранения данных KPI. Наш следующий шаг — разработать сценарий тестирования производительности.

Упражнение

Рассмотрите тест производительности для вашей платформы API. Какова будет цель вашего теста производительности? Проработайте следующие вопросы:

- Какой тип тестирования вы бы провели: стрессовое, нагрузочное или стабильности?
- Что вы хотите измерить: время отклика, доступность, пропускную способность (и т. д.)?
- Какие KPI вы хотите отслеживать?

После этого сформулируйте цель тестирования производительности.

10.1.4. Создание пользовательского потока (user flow)

Теперь мы можем перейти к тому, что именно будет делать наш сценарий тестирования производительности, то есть спланировать, как он будет выполняться. Простого создания сценария, который перечисляет все конечные точки нашего API, а затем запускает определенный объем нагрузки на них, будет недостаточно (хотя он поможет, если мы только хотим понять, как наша система справится с DDOS-атакой). Проблема в том, что полученные результаты вряд ли будут соответствовать реальному сценарию использования нашего продукта, поэтому основанные на них решения будут бесполезными или вредными.

Чтобы построить сценарий, который дает актуальную и ценную информацию, мы должны помнить об одном из основных принципов тестирования производительности:

> Наш тест производительности должен отражать реальное поведение пользователей в рабочей среде настолько точно, насколько это возможно.

То есть необходимо создать сценарий тестирования производительности, который будет максимально близко имитировать поведение пользователей. Хотя мы

не всегда сможем в точности повторить шаги пользователей и их взаимодействие с системой, но стремление воспроизвести их как можно более точно поможет получить более точные результаты. Нам необходимо проанализировать, как пользователи взаимодействуют с нашими системами. При отсутствии данных о пользователях анализировать придется наши ожидания. На основе этой информации определяются пользовательские потоки. Чтобы лучше понять, что подразумевается под пользовательским потоком, давайте рассмотрим пример создания бронирования администратором в нашей песочнице API (рис. 10.1).

```
User flow:          Admin makes a booking
Description:        An admin loads up the report view and makes a booking
Virtual users:      2
Injection profile:  Ramp up 1 minute (1 admin per minute)
Duration:           31000 (Deviance 9500)

Step    Action                          Test Data                 System time (ms)    User think time (ms)
1       Admin loads login page          None                      2000
2       Admin logs into site            None                                          5000 (Deviance 2500)
3       System logs admin into site     None                      2000
4       Admin clicks on report          None                                          2000 (Deviance 1000)
5       System returns report           <Rooms>, <Bookings>       2000
6       Admin loads up booking form     None                                          2000 (Deviance 1000)
7       System returns room details     <Rooms>                   2000
8       Admin completes booking         None                                          10000 (Deviance 5000)
9       System confirms booking         None                      2000

<Rooms>
| roomNumber | roomPrice | roomId | type | image | features | description | accessible |

<Bookings>
| bookingid | checkin | checkout | depositpaid | email | firstname | lastname | phone | roomid |
```

Рис. 10.1. Пользовательский поток, отражающий процесс создания бронирования администратором

Пользовательский поток можно разделить на три части:

- описание пользовательского потока и сведения о нагрузке;
- шаги пользовательского потока;
- требования к данным пользовательского потока.

Пример материала для тестирования производительности

Чтобы помочь вам изучить тестирование производительности, в наш репозиторий GitHub добавлена подборка примеров пользовательских потоков, сценарий тестирования производительности и пример результатов тестирования. Вы можете посмотреть примеры диаграмм пользовательских потоков по ссылке: http://mng.bz/lREj.

Описание пользовательского потока и сведения о нагрузке

Описание содержит название и параметры пользовательского потока, а также сведения о нагрузке — количестве виртуальных пользователей в сценарии тестирования производительности. В нашем примере пользовательского потока два виртуальных пользователя. Это очень низкое значение, но мы ожидаем, что наш сценарий тестирования производительности будет состоять из многих пользовательских потоков. Пользователи взаимодействуют с нашей системой различными способами, а значит, мы должны эмулировать реальную ситуацию со множеством разных потоков. Таким образом, хотя в отдельном потоке количество виртуальных пользователей может быть низким, нагрузка будет расти по мере добавления новых пользовательских потоков.

С помощью профиля нагрузки (*injection profile* на рис. 10.1) мы определяем, как именно хотим ее применить в нашем тесте. То, как пользователи заходят на наши сайты, зависит от контекста, и не всегда все заходят одновременно. Мы можем использовать профиль «большой взрыв», при котором вся нагрузка будет добавлена одновременно. Или можно увеличивать нагрузку со временем — быстро или медленно. В нашем примере профиля, поскольку количество виртуальных пользователей невелико, применен короткий период нарастания нагрузки — ежеминутно добавляется один виртуальный пользователь.

Наконец, значение длительности (*duration*) определяет расчетное время выполнения одного цикла пользовательского потока с применением случайного отклонения (*deviance*, подробнее об этом ниже). Мы можем использовать эту оценку, чтобы задать время выполнения теста производительности. При запуске теста можно настроить пользовательские потоки на бесконечный цикл, пока мы вручную не выключим тест. Или можно остановить его после завершения одного цикла пользовательского потока. Если мы выберем второй вариант, то продолжительность вместе с профилем нагрузки определят, сколько времени должен занять один цикл пользовательского потока. В нашем примере мы ожидаем, что он будет выполняться от 1 минуты 31 секунды до 1 минуты 45 секунд. Учитывая, что это довольно короткое время для проведения теста производительности, предпочтительно запустить пользовательские потоки на 10 минут, а затем завершить сценарий вручную.

Шаги пользовательского потока

Шаги в пользовательском потоке позволяют зафиксировать каждое *действие* (*Action*), которое мы ожидаем от пользователя, а также то, как будет вести себя система. Вот один из шагов пользователя из нашего примера:

Администратор заканчивает бронирование (*Admin completes booking*).

Этот шаг описывает наши ожидания от пользователя, но не содержит подробностей. Это позволяет отразить в пользовательских потоках ожидания как технических, так и бизнес-команд. Шаг дает достаточно информации о том, что мы должны отразить в сценарии тестирования производительности. В случае приведенного в качестве примера шага нам нужно отправить HTTP-запрос `POST /booking/`. Но его описание не настолько подробно, чтобы затруднять чтение или поддержку.

Каждый шаг содержит подробную информацию о том, какие *тестовые данные* (*Test Data*) необходимы. Дополнительные данные важны в тестировании производительности, помогая сэкономить время при создании сценария тестирования. Чтобы таблицу шагов было легко читать, в нее включены ссылки на требования к данным, которые подробно описаны под таблицей. Так, например, для шага

Система возвращает информацию о номере (*System returns room details*)

требуются тестовые данные `<Rooms>`. В этом столбце можно указывать данные, которые должны быть в системе при запуске сценария, или тестовые данные, которые будет использовать сценарий тестирования производительности. Во втором случае их можно сохранить в файлах формата .csv или подобных.

В столбцах *время ожидания системы* (*System time*) и *время ожидания пользователя* (*User think time*) указаны наши оценки того, сколько времени потребуется приложению для ответа и сколько времени потребуется пользователю для отправки запроса. Время ожидания системы для каждого шага составляет 2000 миллисекунд, поскольку мы пока не знаем, как долго система будет отвечать. Поскольку мы хотим, чтобы наши сценарии были репрезентативными, нужно добавить время ожидания, которое похоже на поведение пользователя. Ни один пользователь не сделает все запросы, необходимые для цикла, один за другим менее чем за 10 миллисекунд, поэтому такое значение в сценарии приведет к неточным результатам. Вместо этого мы хотим учесть время, которое необходимо пользователю (или другим приложениям, в зависимости от контекста) для формирования запроса и его отправки. Поэтому мы указываем расчетное время и время отклонения (deviance):

5000 (Deviance 2500),

ожидая, что пользователю потребуется от 2,5 до 7,5 секунды для отправки запроса. Такой разброс объясняется тем, что разным пользователям требуется различное время. Таким образом, ни один тест производительности не будет действительно одинаковым, однако это добавляет реализма.

Требования к данным пользовательского потока

В конце описания пользовательского потока мы можем разместить подробные требования к данным. Названия наборов данных указываются как ссылки в таблице шагов. Это удобно для планирования требуемых тестовых данных и дает представление о том, как их генерировать. При создании `<Bookings>` я использовал инструмент Mockaroo (https://www.mockaroo.com/), который предлагает различные наборы случайных данных. Например, если мне нужен номер телефона для бронирования, я могу использовать рандомизатор телефонных номеров в Mockaroo.

Когда у нас есть готовый пользовательский поток, мы должны конкретизировать требования, например:

- порядок выполнения пользовательского потока;
- примерное время выполнения цикла пользовательского потока для определения продолжительности теста производительности;
- какие данные нам требуются, когда приложение настроено для проведения теста производительности;
- требования к входным данным, когда мы запускаем сценарий тестирования производительности, например данные бронирования.

Все это необходимо для сценария тестирования производительности, но настоящие преимущества пользовательских потоков заключаются в том, что этот подход позволяет нам планировать и выполнять тесты производительности, пока разрабатывается производственный код. Кроме того, эти описания могут использоваться неоднократно.

В идеале сценарии тестирования производительности должны совершенствоваться по мере развития продукта. Анализ пользовательских потоков помогает точнее спланировать тесты производительности и адаптировать их к изменениям продукта. Вместо того чтобы ждать окончания разработки, мы можем описывать пользовательские потоки по мере развития системы. Например, представьте, что API нашей песочницы еще только создается и пока разработана только функция бронирования. Мы опишем пользовательский поток для бронирования и выполним тест производительности на его основе. Когда будет создана функция брендинга, мы можем создать пользовательский поток для брендинга и добавить его в сценарий тестирования производительности. Это позволяет использовать последовательный подход к тестированию производительности, при котором мы реагируем на изменения по мере создания нашей системы и получения новой информации.

Упражнение

Сценарий тестирования производительности состоит из нескольких пользовательских потоков. Попробуйте создать новый сценарий пользовательского потока для процесса бронирования, в котором гость заходит на сайт, находит номер и бронирует его. Вы можете сравнить свой пользовательский поток с примерами потоков в репозитории книги (http://mng.bz/WM1x).

10.2. ВЫПОЛНЕНИЕ ТЕСТА ПРОИЗВОДИТЕЛЬНОСТИ

Когда пользовательский поток создан, можно добавить его в сценарий тестирования производительности. Рассмотрим необходимые для этого шаги на примере пользовательского потока, имитирующего создание бронирования администратором.

10.2.1. Настройка инструментов для тестирования производительности

Мы будем использовать Apache JMeter — инструмент тестирования производительности с открытым исходным кодом. Мы выбрали JMeter, поскольку он поддерживается уже более 20 лет (первый релиз был выпущен в 1998 году) и содержит все инструменты, необходимые для создания теста производительности:

- HTTP-семплеры для выполнения запросов;
- управление переменными;
- управление файлами cookie и заголовками HTTP;
- логические контроллеры;
- загрузчики данных CSV.

Мы воспользуемся этими инструментами при создании сценария.

Распределенная установка

Одной из проблем при тестировании производительности являются требования к ресурсам для создания нагрузки. Тест производительности будет использовать значительные ресурсы процессора, RAM и сетевого обмена. Если ресурсов недостаточно, это может повлиять на результаты. Например, если тест производительности достиг максимума сетевого обмена, мы можем начать регистрировать

большие значения времени отклика, что связано не с тестируемой системой, а с нашими средствами тестирования производительности.

Для решения проблемы ресурсов мы можем использовать возможность JMeter по запуску сценариев тестирования производительности в распределенной архитектуре, как показано на рис. 10.2.

Распределенная установка позволяет подключить несколько jmeter-серверов, которые соединены с экземпляром JMeter. Обычно каждый jmeter-сервер разворачивается на собственном сервере, отдельно от других, а затем подключается к основному приложению JMeter по сети. Это позволяет запустить сценарий тестирования производительности из нашего основного экземпляра JMeter, который затем отправляет инструкции каждому экземпляру jmeter-сервера для создания нагрузки — отправки запросов к тестируемой системе. Каждый jmeter-сервер, получив ответ, отправит информацию главному экземпляру JMeter для сохранения результатов. Таким образом, мы можем создать тестовый комплекс, который генерирует нагрузку, предотвращая перегрузку ресурсов и обеспечивая точность результатов.

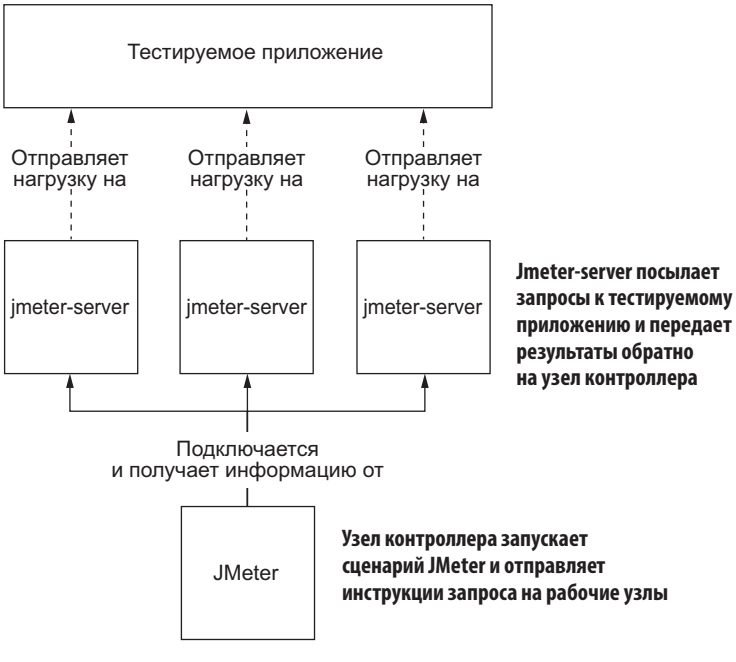

Рис. 10.2. Модель распределенной тестовой установки JMeter со связями между рабочими узлами и узлами контроллера

Работа с распределенными установками

При использовании jmeter-сервера мы должны помнить о нескольких вещах. Распределенный метод работает за счет того, что основной экземпляр JMeter отправляет копию сценария тестирования производительности каждому подключенному экземпляру jmeter-сервера. Это означает, что когда мы устанавливаем количество виртуальных пользователей для определенного потока в сценарии тестирования производительности, мы должны разделить его на число экземпляров jmeter-сервера, которые будем использовать. Также необходимо убедиться, что скопированы все вспомогательные файлы данных, такие как CSV-файлы, потому что jmeter-сервер будет искать их локально, а не через сетевое соединение с основным приложением JMeter.

Подробнее о том, как работает распределенная установка и как ее создать, вы можете узнать из документации Apache JMeter (http://mng.bz/BZ5v). А сейчас давайте установим и запустим JMeter.

Установка

Как упоминалось ранее, JMeter — это приложение с открытым исходным кодом, которое можно бесплатно загрузить по адресу http://jmeter.apache.org/download_jmeter.cgi. Нам потребуется бинарная версия, поэтому загрузите файл формата .zip или .tgz и потом извлеките содержимое из архива. Затем переместите папку в место, где вы предпочитаете хранить свои приложения.

Когда JMeter распакован, нужно просто запустить приложение, открыв папку JMeter, а затем папку bin (это можно сделать в командной строке/терминале или через Explorer, Finder и пр.). В Windows мы должны запустить `jmeter.bat`, а в Mac или Linux — `jmeter.sh`. После запуска сценария `jmeter` нам будет представлен пустой план тестирования, который мы начнем заполнять для создания нашего сценария тестирования производительности.

10.2.2. Создание сценария тестирования производительности

Теперь мы готовы к созданию собственного сценария тестирования производительности, начиная с настройки плана тестирования (Test Plan) и групп потоков (Thread Groups).

План тестирования и группы потоков

План тестирования JMeter содержит все, что будет выполнять наш сценарий. Кликните по тестовому плану в левой панели, чтобы вызвать его детали в правой панели. Здесь мы изменим название плана тестирования на *Example performance test script* (пример сценария тестирования производительности), а также добавим две пользовательские переменные, как показано на рис. 10.3:

- Name: server, Value: localhost
- Name: port, Value: 80

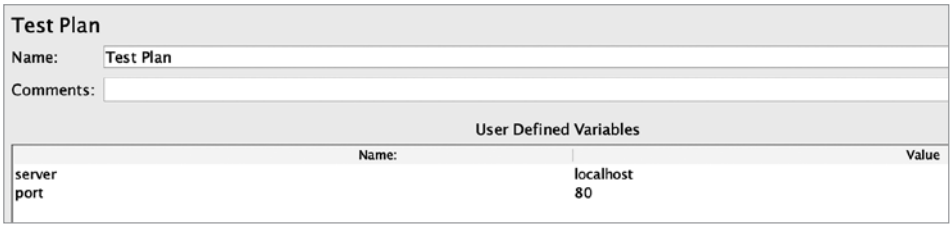

Рис. 10.3. Элемент JMeter Test Plan

Мы скоро вернемся к переменным, а сейчас начнем добавлять группы потоков в план тестирования.

Группы потоков отвечают за организацию пользовательских потоков и настройку нагрузки. Как мы обсуждали ранее, сценарий тестирования производительности будет иметь несколько пользовательских потоков, выполняемых одновременно, чтобы эмулировать множество способов, которыми пользователи будут взаимодействовать с нашим приложением. Соответственно, в плане тестирования JMeter может быть много групп потоков, каждая из которых будет содержать определенный пользовательский поток.

Более подробно о том, как настраивать группы потоков, мы расскажем позже. Пока нам нужно только создать группу потоков, кликнув правой кнопкой мыши на плане тестирования в левой панели, чтобы вызвать меню, а затем выбрав Add > Threads > Thread Group. Затем мы дадим ей имя, кликнув на группе потоков и изменив имя на *Admin makes a booking* (администратор осуществляет бронирование), как показано на рис. 10.4.

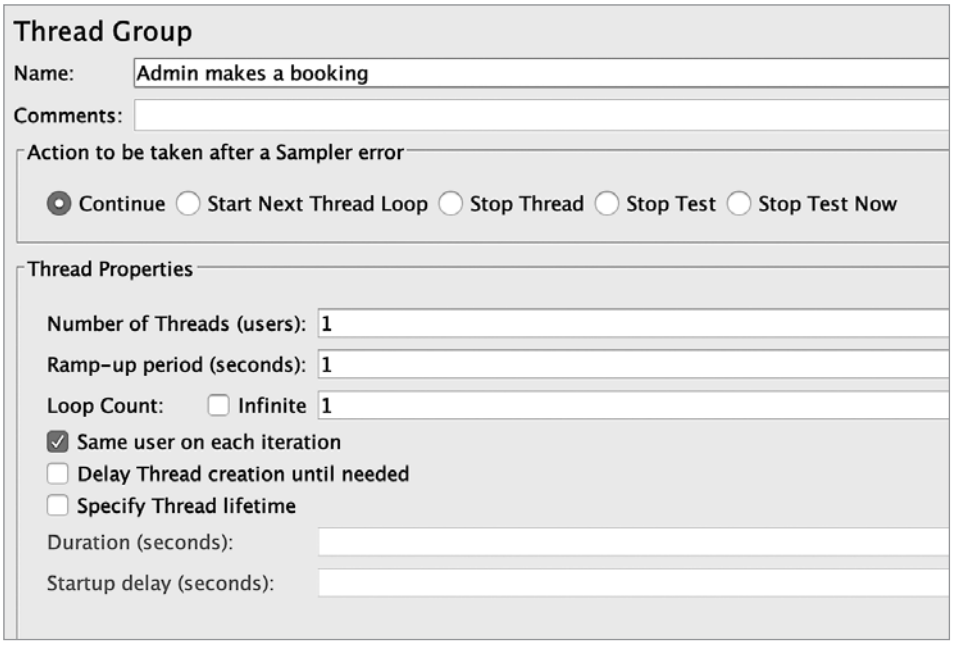

Рис. 10.4. Элемент JMeter Thread Group

Добавление элементов в JMeter

В этом примере мы добавим в план тестирования множество элементов. Для наглядности можно добавить в план тестирования JMeter такие элементы, как HTTP-семплеры, логические контроллеры и элементы Config, либо выделив элемент в левой панели и выбрав Edit > Add в главном меню, либо кликнув правой кнопкой мыши на элементе в левой панели и выбрав Add. В любом случае ваш элемент будет добавлен в качестве дочернего.

Далее нам нужно добавить запросы, которые эмулируют вход администратора в приложение и последующее бронирование. Чтобы структурировать эти два действия, мы создадим два простых логических контроллера для хранения HTTP-запросов. Простой логический контроллер располагается в Add >Logic Controller > Simple Controller. Создав контроллеры, переименуем их, как показано на рис. 10.5:

- *Log into admin* (войти в систему под именем администратора);
- *Admin makes booking* (администратор осуществляет бронирование).

Это дает нам структуру, необходимую для начала добавления HTTP-запросов.

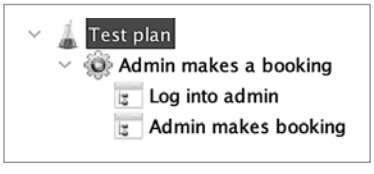

Рис. 10.5. План тестирования JMeter и добавление простых контроллеров в группу потоков Admin makes a booking

Менеджер http-заголовков (http header manager)

Прежде чем мы начнем добавлять HTTP-запросы, упростим наш план, создав менеджер HTTP-заголовков непосредственно под группой потоков (не внутри простых контроллеров). Длят этого перейдем по пути Add > Config Element > HTTP Header Manager. Менеджер HTTP-заголовков позволяет настроить, какие HTTP-заголовки будут добавлены к HTTP-запросам в нашей группе потоков. Мы добавим в HTTP Header Manager следующие HTTP-заголовки, как показано на рис. 10.6:

- Host: `${server}:${port}`
- Accept: `application/json`
- Accept-Language: `en-GB,en;q=0.5`
- Accept-Encoding: `gzip, deflate, br`
- Content-Type: `application/json`

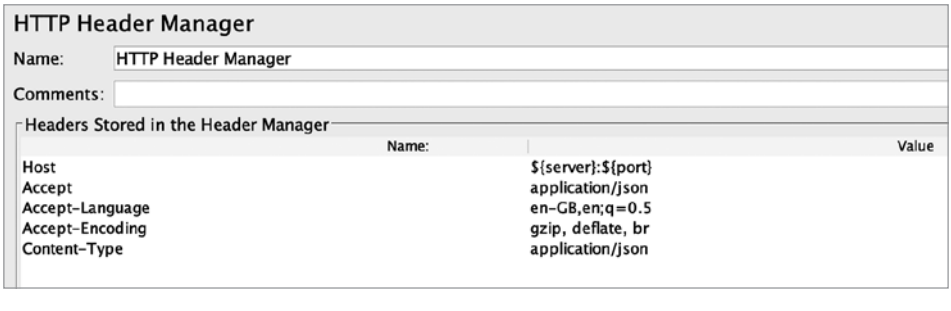

Рис. 10.6. Элемент JMeter HTTP Header Manager

Эти заголовки будут добавляться автоматически в каждый запрос, который мы будем отправлять. Таким образом, не нужно добавлять HTTP-заголовки к каждому запросу, если мы этого не хотим.

Кроме того, обратите внимание, что заголовок Host, который мы добавили, имеет значение `${server}:${port}`. Это переменные JMeter, и они соответствуют переменным сервера и порта, которые мы задали в плане тестирования. Поскольку мы установили переменную сервера на `localhost`, а переменную порта на `80`, при выполнении нашего скрипта запись `${server}:${port}` превратится в `localhost:80`. Использование переменных облегчает обновление сценария для работы в различных средах.

Контроллер login into admin

Когда менеджер HTTP-заголовков настроен, давайте добавим HTTP-запрос в контроллер `Login into admin`, выбрав `Add > Sampler > HTTP Request`. HTTP-запрос настраивается в семплере HTTP Request — задаются такие параметры, как методы HTTP, URI, тело запроса и т. д.

Наш первый запрос эмулирует попытку администратора проверить, вошел ли он в систему (разумеется, он не в системе), поэтому мы добавляем следующие данные:

- Name: `POST /auth/validate`
- Server name or IP: `${server}`
- Port number: `${port}`
- HTTP Request: `POST`
- Path: `/auth/validate`

Как показано на рис. 10.7, также нужно выбрать `Body Data` и добавить пустой объект `{}`, потому что у нас нет маркера для отправки на проверку.

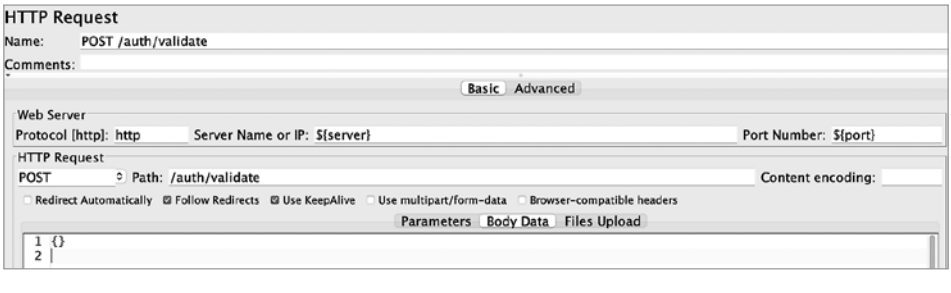

Рис. 10.7. Элемент JMeter HTTP Request

Далее добавим HTTP-запрос, необходимый для входа в систему:

- Name: `POST /auth/login`
- Server name or IP: `${server}`
- Port number: `${port}`
- HTTP Request: `POST`
- Path: `/auth/login`
- Body Data:

```
{
    "username":"admin",
    "password":"password"
}
```

Затем добавим запрос, загружающий счетчик сообщений:

- Name: `GET /message/count`
- Server name or IP: `${server}`
- Port number: `${port}`
- HTTP Request: `GET`
- Path: `/message/count`

И наконец, запрос на предоставление комнат:

- Name: `GET /room/`
- Server name or IP: `${server}`
- Port number: `${port}`
- HTTP Request: `GET`
- Path: `/room/`

Тестирование производительности API GraphQL

Мы можем использовать HTTP-семплеры для отправки GraphQL-запросов, обновив данные тела, чтобы они содержали GraphQL-запрос, а не просто JSON-объект. Несколько сложнее разбирать ответы, но к счастью, JMeter имеет широкие возможности по работе с GraphQL.

Мы создали все запросы, необходимые для эмуляции входа администратора в наше приложение. Но нам осталось добавить еще один элемент к `POST /auth/validate` и `GET /room/` — Constant Timers (постоянные таймеры).

Добавляя постоянный таймер через Add > Timers > Constant Timer в качестве дочернего элемента к запросу, мы можем имитировать время ожидания пользователя, которое указано в описании пользовательского потока. Например, если мы добавим постоянный таймер к `POST /auth/validate` и в поле Thread Delay укажем `${__Random(2500,7500)}`, то скрипт будет ждать от 2,5 до 7,5 секунды перед выполнением запроса (рис. 10.8).

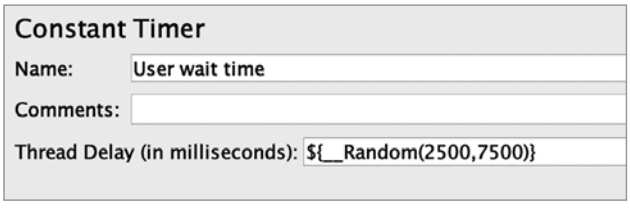

Рис. 10.8. Элемент JMeter Constant Timer

Чтобы завершить этот раздел сценария, добавим еще один постоянный таймер в `GET /room/` с задержкой потока `${__Random(1000,3000)}`, что закончит настройку первого простого контроллера.

Контроллер admin makes a booking

Далее обратимся к контроллеру `Admin makes a booking`. Мы хотим имитировать получение отчета с доступными бронированиями, просмотр деталей номера, а затем бронирование. Начнем с добавления двух GET-запросов, первый из которых — для `/report/`:

- Name: `GET /report/`
- Server name or IP: `${server}`
- Port number: `${port}`
- HTTP Request: `GET`
- Path: `/report/`

Второй запрос — для `/room/`:

- Name: `GET /room/`
- Server name or IP: `${server}`

- Port number: `${port}`
- HTTP Request: `GET`
- Path: `/room/`

Остается последний запрос, который создает бронирование в системе. Для него необходимо добавить некоторые дополнительные элементы. Начнем с создания HTTP-запроса со следующими данными:

- Name: `POST /booking/`
- Server name or IP: `${server}`
- Port number: `${port}`
- HTTP Request: `POST`
- Path: `/booking/`

Далее выберем опцию Body Data и добавим следующий объект JSON:

```
{
    "depositpaid": ${depositpaid},
    "firstname": "${firstname}",
    "lastname": "${lastname}",
    "roomid": 2,
    "bookingdates": {
        "checkin": "${checkin}",
        "checkout": "${checkout}"
    }
}
```

Обратите внимание, что большинство значений в этом объекте являются переменными JMeter (например, `${depositpaid}`), а не жестко заданными значениями. И снова причина заключается в том, что мы хотим эмулировать реальное поведение пользователя, а реальные администраторы или гости не будут добавлять одно и то же бронирование несколько раз (к тому же в приложении есть ограничения, не позволяющие дублировать бронирования).

Поэтому нам нужно убедиться, что каждый раз, когда в потоке вызывается запрос `POST /booking`, отправляются новые данные бронирования. Мы можем сделать это, управляя данными запроса с помощью CSV-файла, настроив запрос на выбор новой строки данных из CSV при каждом запуске. Таким образом, сначала нам нужно создать CSV-файл с данными, которые выглядят примерно так:

```
Silvie,Alyonov,salyonov0@wikispaces.com,92511364701,false,2020-01-01,2020-01-02
Ambros,Eary,aeary1@ox.ac.uk,41789748281,true,2020-01-03,2020-01-04
```

Каждая строка содержит случайно сгенерированное имя, фамилию, адрес электронной почты, номер телефона, размер депозита, даты заселения и выселения. Обычно CSV содержит сотни или тысячи таких строк данных, генерируемых, например, с помощью комбинации из Mockaroo и Excel.

После того как мы создали CSV-файл и сохранили его там же, где размещена папка JMeter, добавим элемент CSV Data Set Config в контроллер `Admin makes a booking`, выбрав Add > Config Element > CSV Data Set Config. Этот элемент конфигурации позволяет импортировать наш CSV-файл и объявить переменные для использования. В нашем элементе CSV Data Set Config зададим следующие параметры (рис. 10.9):

- Filename: `./bookings.csv` (имя файла `.csv`)
- File encoding: `utf-8`
- Variable names: `firstname,lastname,email,phone,depositpaid,checkin,checkout`

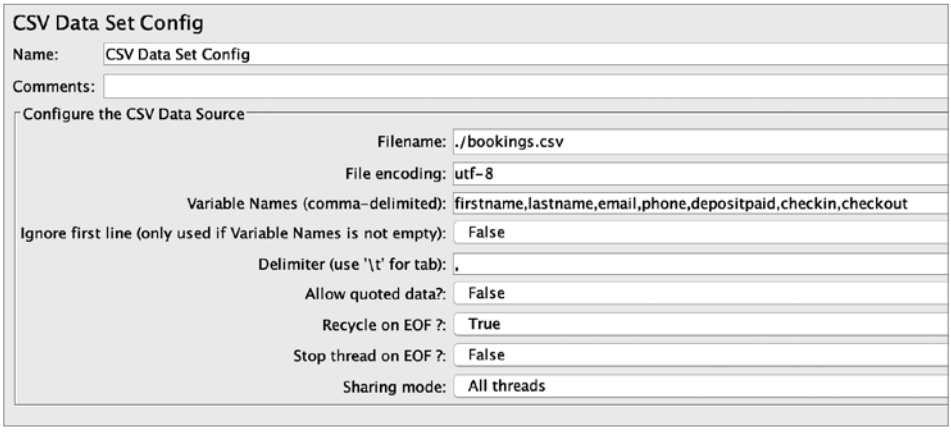

Рис. 10.9. Элемент JMeter CSV Data Set Config

Обратите внимание, что имена полей в Variable Names совпадают с названиями колонок в CSV-файле и что имена переменных соответствуют переменным, используемым в объекте Body Data. Например, если при выполнении теста производительности первая строка CSV-файла имеет имя `Silvie`, то первое бронирование в этой группе Thread Group добавит `Silvie` в Body Data для запроса `POST /booking/`, а при отправке второго бронирования будет использована

вторая строка — `Ambros`. Это гарантирует, что каждое отправленное бронирование будет отличаться от других.

Наконец, чтобы завершить сценарий, мы добавляем постоянные таймеры для имитации времени ожидания к следующим запросам:

- `GET /room/: ${__Random(2500,7500)}`
- `POST /booking: ${__Random(5000,5000)}`

Это завершает настройку пользовательского потока в нашем сценарии тестирования производительности, но нам еще нужно настроить JMeter так, чтобы он выводил необходимые показатели.

Слушатели

Наш сценарий тестирования производительности уже имеет все необходимые данные для создания нагрузки на веб-API. Однако пока он нигде не сохраняет результаты. Чтобы решить эту проблему, нужно добавить «слушателей». В частности, мы добавим два слушателя непосредственно в план тестирования, чтобы они фиксировали все метрики от каждой группы потоков. Нам потребуются следующие слушатели:

- *Просмотр дерева результатов* (View Results Tree) — слушатель, который показывает подробные результаты каждого отправленного запроса и ответа. Он полезен для отладки сценариев тестирования производительности.
- *Краткий отчет* (Summary Report) — слушатель, который группирует информацию о времени ответа, ошибках и пропускной способности на основе имени HTTP-семплера. Например, если существует множество запросов с именем `POST /booking` в нескольких группах потоков, их метрики хранятся вместе. Это полезно для группировки и экспорта времени отклика по конечным точкам для последующего анализа.

Слушатели добавляются с помощью меню Add > Listener. Как мы увидим, существует большое число типов слушателей, которыми мы могли бы воспользоваться. Однако, учитывая, что нашей целью является получение кодов состояния, будет достаточно краткого отчета. Последнее, что необходимо настроить, — поле имени файла в кратком отчете, нужно добавить имя файла, например `jmeter-results.csv`. Это гарантирует, что при запуске сценария тестирования производительности результаты будут сохранены.

Отключение просмотра дерева результатов

Слушатели просмотра дерева результатов являются отличными инструментами для отладки сценариев тестирования производительности. Они предоставляют много подробной информации по каждому запросу, а также обеспечивают визуализацию успешных и неудачных запросов. Однако такой слушатель является ресурсоемким элементом, и JMeter рекомендует отключать его при выполнении теста производительности. Для этого нам нужно просто щелкнуть по нему правой кнопкой мыши и выбрать Toggle, что приведет к отключению слушателя и закрашиванию его серым цветом в левой панели.

Упражнение

Как мы уже знаем, сценарии тестирования производительности могут содержать несколько потоков пользователей. До сих пор мы настроили один поток для администратора, выполняющего бронирование. Добавьте к нему поток пользователей, который вы создали в предыдущей части главы, и настройте новую группу потоков в плане тестирования JMeter. В случае затруднений ознакомьтесь с примерами потоков или сценарием JMeter, которые находятся в репозитории книги (http://mng.bz/WM1x).

Генеральная репетиция

Поскольку наши сценарии предназначены для воспроизведения сложного поведения пользователей, они, по понятным причинам, сами являются сложными. Поэтому разумно провести генеральную репетицию перед тем, как запустить тест производительности в полном объеме. Генеральная репетиция обычно заключается в запуске сценария тестирования производительности со всеми необходимыми данными, загруженными в сценарий и систему, но только с одним виртуальным пользователем на группу потоков. Запустив сценарий с небольшой нагрузкой, мы можем выявить и устранить проблемы, связанные с тем, что группы потоков мешают друг другу, или с ошибками в конфигурациях. Например, при выполнении генеральной репетиции для примера тестового сценария, представленного в репозитории книги, возникла целая серия из ошибок 400. Оказалось, что одна группа потоков удаляла номера, в которые другая группа потоков пыталась отправить бронирования.

Чтобы избежать головной боли, связанной с провалом сценария тестирования производительности из-за ошибок, проводите генеральную репетицию каждый раз, когда в сценарий вносятся значительные изменения.

10.3. ВЫПОЛНЕНИЕ И ОЦЕНКА ТЕСТА ПРОИЗВОДИТЕЛЬНОСТИ

Теперь, когда наш сценарий тестирования производительности готов, пришло время рассмотреть дополнительные факторы, которые необходимо учесть до его запуска и анализа результатов.

10.3.1. Подготовка и проведение теста производительности

Перед каждым тестом производительности необходимо предпринять несколько шагов, чтобы обеспечить точность результатов. Давайте рассмотрим эти шаги и их значение.

Подготовка тестируемой среды

Если сценарий тестирования производительности имитирует поведение пользователей, то и приложение должно быть развернуто таким образом, чтобы максимально повторять живую среду. Это необходимо для того, чтобы полученные результаты были максимально точными. Например, если приложение развернуто с помощью таких инструментов, как Kubernetes, Docker, Ansible или Puppet, которые используют сложные инфраструктуры с масштабируемыми контейнерами, балансировщиками нагрузки и синхронизацией баз данных, то запуск теста производительности на версии приложения, которая находится на одном сервере с ограниченными ресурсами, вряд ли даст точные результаты. Мы должны убедиться, что среда тестирования производительности соответствует производственной среде настолько близко, насколько это возможно в рамках ограничений на время и бюджет проекта.

Помимо обеспечения соответствия инфраструктуры, также необходимо знать о других элементах, которые делают приложение максимально приближенным к жизни:

- *Наполнение баз данных*. Базы данных обычно не бывают пустыми, когда пользователи обращаются к нашим системам. Поэтому добавление ожидаемого среднего объема данных в базу поможет повысить точность результатов теста.
- *Заполнение кэша*. То же правило применимо и к кэшированию. Пользователи не всегда попадают в систему без кэша, поэтому заполнение кэша с помощью инструмента или вручную также будет способствовать увеличению точности.

- *Цикличность/загруженность очередей*. Порядок запланированных событий в системе должен быть разумным, например, чтобы не включалась одновременная обработка сразу всех очередей.

Это лишь несколько примеров того, как нужно подготовить приложение к тестированию производительности. Обсудив требования к тестированию производительности с командой, мы можем узнать больше важной информации.

Оповещение заинтересованных сторон

Представьте, что мы находимся на этапе запуска теста производительности: инфраструктура среды создана, приложение развернуто и настроено. Мы запускаем тест, но обнаруживаем, что кто-то развернул обновление среды или проводит тестирование и изменил данные, на которые мы полагаемся. Это выбивает из колеи и приводит к потере времени. Именно поэтому следует заранее уведомить о наших планах всех коллег, кто имеет доступ или претендует на среду, в которой мы проводим тестирование производительности. Информирование каждого из них о том, когда мы собираемся провести тест производительности, сколько примерно времени он займет и что мы дополнительно сообщим о его завершении, поможет избежать неудач, которые приводят к повторному тестированию или, что еще хуже, к неточным данным.

Окончательная установка и выполнение

Когда все настроено в среде и все оповещены, можно приступить к работе. Все, что осталось сделать, — настроить систему тестирования производительности и начать тестирование. Этот этап может включать в себя следующие шаги:

- *Настройка распределенной системы*. Как мы обсуждали выше, JMeter способен работать в распределенной конфигурации. Мы можем использовать несколько jmeter-серверов для создания нагрузки от центрального экземпляра JMeter. В этом случае нужно создать экземпляры jmeter-сервера и настроить JMeter для подключения к ним.
- *Проверка файлов данных тестирования*. Следует убедиться, что все необходимые CSV-файлы находятся в месте, где их найдет сценарий JMeter. Файлы данных должны быть проверены на актуальность и, при необходимости, скопированы на распределенные jmeter-серверы.
- *Включение инструментов мониторинга*. Хотя наш инструмент тестирования производительности будет собирать данные по каждому отправленному запросу, нам, скорее всего, потребуются и другие показатели. Это требует

настройки инструментов для мониторинга KPI, например показателей системы, сервера и базы данных, и обеспечения сбора информации для будущего анализа.

- *Настройка групп потоков*. Группы потоков необходимо настроить таким образом, чтобы они генерировали желаемую нагрузку. Например, для группы потоков `Admin creates a booking` нужно обновить следующие поля:
  - число потоков (Number of Threads): 2;
  - период наращивания (Ramp-up period): 120 секунд;
  - количество циклов (Loop Count): бесконечное (чтобы мы могли вручную завершить сценарий тестирования производительности через заданное время).
- *Обновление среды*. Мы задали параметры сервера и порта для сценария JMeter, чтобы быстро обновлять их в плане тестирования при указании на конкретную среду. Нужно проверить, содержат ли они актуальные данные, и если это не так, обновить их.

Убедившись, что все настроено, можем приступить к тестированию. Но следует помнить, что JMeter нужно запускать в режиме headless, а не UI. Такова рекомендация команды JMeter, поскольку запуск в режиме пользовательского интерфейса требует больших ресурсов и может повлиять на точность результатов. Итак, воспользуемся командной строкой:

```
jmeter -n -t ./example-performance-test.jmx -l ./perfstats.csv
```

Эта команда запустит сценарий тестирования производительности в режиме CLI и выведет метрики в файл perfstats.csv. Когда придет время завершить тест, мы сделаем это вручную, закроем инструменты мониторинга и соберем показатели для дальнейшего анализа.

Упражнение

Запуск теста производительности с нашим примером сценария тестирования производительности на полностью развернутом экземпляре платформы restful-booker с распределенной настройкой JMeter сложен для первого опыта. Для начала, чтобы понять, как работают тесты производительности, и получить примеры результатов, протестируйте API песочницы локально. При этом попробуйте запустить JMeter в режиме CLI в течение 5 минут вместе с инструментом для измерения нагрузки на ЦП вашей системы.

10.3.2. Анализ результатов

После завершения тестов наступает очередь, пожалуй, самой трудной части — анализа результатов тестирования производительности для выявления проблем, требующих диагностики. Сложность состоит в том, что измеренные показатели в основном отображают только симптомы, а не саму проблему. Добавьте к этому многочисленные способы, которыми можно повлиять на производительность приложения. Тем не менее существует несколько способов сопоставления данных сценария тестирования производительности с показателями KPI, чтобы выделить области для дальнейшего исследования.

Измерение времени отклика

Время отклика определяется с учетом количества пользователей, подключенных в определенный момент времени к системе. Если приложение фиксирует большое время отклика, следует проверить серверные и сетевые KPI, чтобы определить, потребляло ли приложение слишком много ресурсов в тот момент.

Пропускная способность и емкость

Показатели, сколько данных или транзакций было обработано в течение всего теста, измеряются с учетом числа подключенных пользователей. Они показывают, какой объем данных может обработать приложение за определенное время. Также можно оценить, насколько быстро приложение может обработать определенный объем данных, и выявить снижение пропускной способности. Последнее может привести к тому, что пользователи не смогут подключиться к системе.

Мониторинг kpi: сеть

Сетевые KPI можно использовать для измерения объема трафика, поступающего в приложение и выходящего из него. Если исходящий трафик превышает входящий, возможно, есть проблемы с кэшированием или отправкой объемных файлов.

Мониторинг kpi: сервер

Некоторые KPI могут служить для диагностики проблем и измерения производительности приложения. Например, если серверные KPI использования процессора и памяти показывают постепенное увеличение или резкое снижение производительности, это может указывать на различные проблемы.

Отчет об ошибках

Фиксация кодов состояния, таких как 404, 503 и 500, помогает определить ситуации, в которых сервер может стать недоступным. Мы можем объединить эту информацию с другими KPI, чтобы понять состояние системы, в котором начинают выдаваться коды ошибок.

Это подводит нас к анализу результатов проведенного теста производительности. Для начала давайте вспомним наши требования к производительности:

> Доступность должна быть выше 95 %, когда к системе подключено 40 виртуальных пользователей.

Чтобы определить, соответствует ли наше приложение этим требованиям, построим график процентного соотношения числа кодов ошибок в последних 30 запросах и на протяжении всего теста (рис. 10.10).

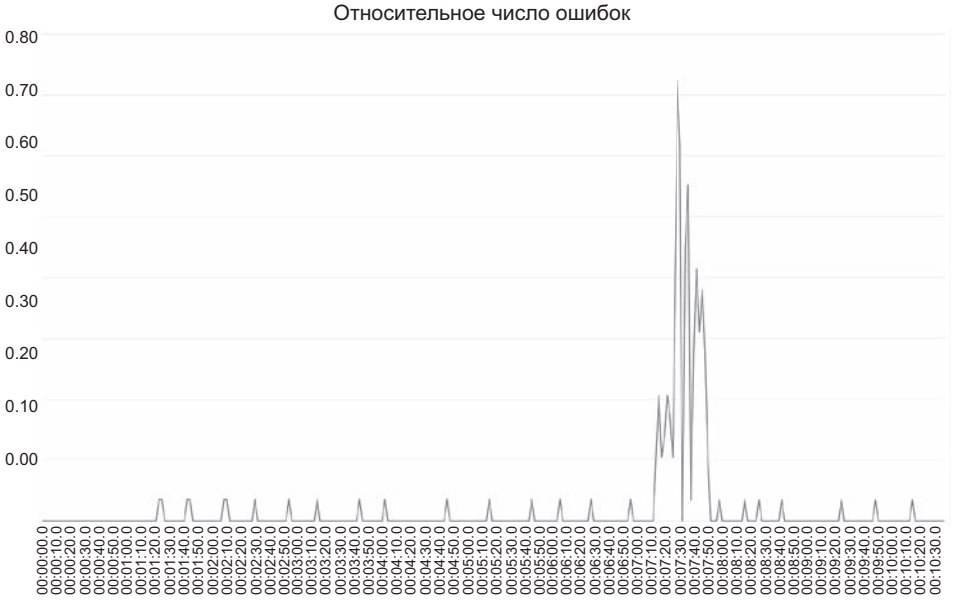

Рис. 10.10. График динамики времени отклика за период проведения теста производительности

Мы также можем построить график загрузки процессора (ЦП) нашими веб-API во времени (рис. 10.11).

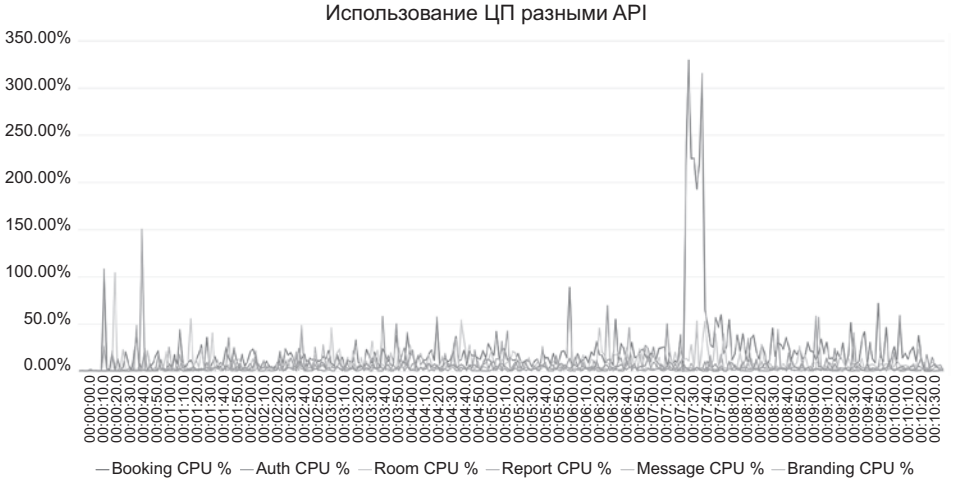

Рис. 10.11. График использования ЦП разными API за время проведения теста производительности

Как мы видим, в районе отметки 7,5 минуты наблюдается значительный скачок, причем как во времени отклика, так и в использовании `booking` API ресурсов ЦП (использование ЦП является бóльшим из двух всплесков), что более наглядно показано на рис. 10.12.

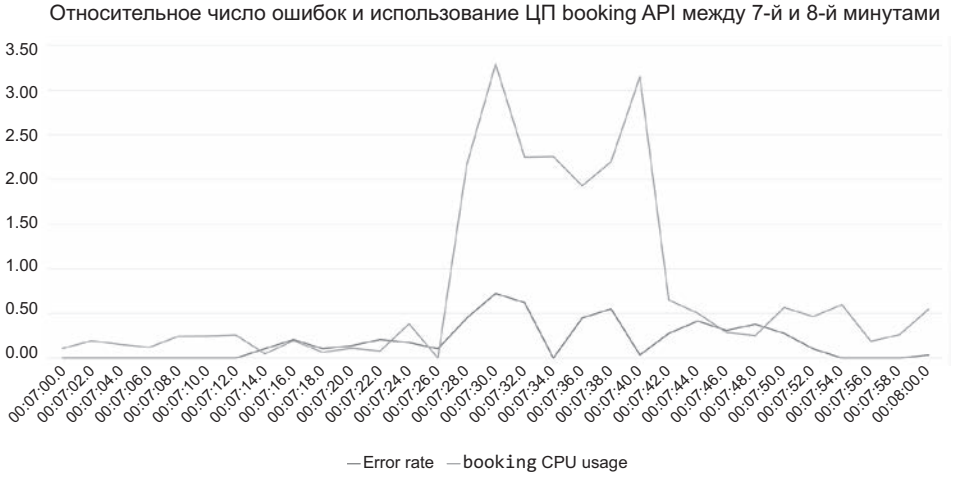

Рис. 10.12. Сопоставление информации о времени отклика и использовании ЦП

Таким образом, когда `booking` API резко увеличивает использование ЦП при нагрузке, API начинает ошибаться. Это означает, что требования к производительности не удовлетворяются. Команде необходимо более детально изучить `booking` API, чтобы определить причину высокой загрузки процессора и устранить ее.

Начинаем сначала

В этой главе мы рассмотрели многие аспекты тестирования производительности, требующие планирования. И хотя для подготовки первого тестирования необходимы определенные инвестиции, возможности повторного использования описаний пользовательских потоков и сценариев тестирования производительности позволяют постепенно расширять наши тесты параллельно с развитием приложения, что означает более быстрое получение результатов.

10.4. ОЖИДАНИЯ ОТ ТЕСТИРОВАНИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Тестирование производительности как концепция показывает, как мы можем взглянуть на продукты с разных точек зрения. Независимо от того, проводим ли мы тестирование, чтобы узнать о производительности продукта, или к нему предъявляются явные требования, мы все равно работаем с одним и тем же продуктом. Меняется только наше отношение и мотивация в отношении того, что мы надеемся узнать. Поэтому в модели стратегии тестирования тестирование производительности охватывает область реализации (рис. 10.13), подобно исследовательскому и автоматизированному тестированию.

Рис. 10.13. Модель стратегии тестирования демонстрирует, что тестирование производительности фокусируется на области реализации

Со временем соображения о том, какие риски необходимо снизить, могут измениться. Как упоминалось ранее, это может быть обусловлено изменением представлений пользователей о качестве. Продукт, имеющий нужные функции, но плохо работающий, не будет востребован. Поэтому если соответствующая характеристика качества важна для пользователей, мы должны как можно раньше начать активно изучать производительность наших продуктов.

ИТОГИ

- Тестирование производительности является способом получения информации о продукте, который отличается от других видов тестирования.
- К распространенным типам тестирования производительности относятся тестирование базовой версии, нагрузочное, стрессовое и тестирование стабильности.
- Мы можем собирать данные для определения доступности приложения, времени отклика, пропускной способности и использования ресурсов.
- Для успешного анализа результатов необходимо определить четкую цель тестирования производительности.
- Ключевые показатели производительности помогают понять, как ведет себя система во время тестирования, и устранить выявленные проблемы.
- Мы можем отслеживать низкоуровневые, серверные и KPI базы данных.
- Сценарий тестирования производительности должен воспроизводить действия пользователей настолько близко, насколько это возможно.
- Мы можем спланировать воспроизведение поведения пользователей с помощью документов пользовательского потока.
- Apache JMeter — инструмент с открытым исходным кодом, который содержит все необходимые функции для создания сценария тестирования производительности.
- При создании сценария тестирования производительности в JMeter нужно добавлять групповые потоки, семплеры HTTP-запросов, логические контроллеры, постоянные таймеры и элементы конфигурации.
- При тестировании производительности условия работы приложения должны быть максимально приближены к реальным.
- Мы сравниваем результаты теста производительности с данными KPI, чтобы выявить проблемы, требующие решения.

11 Тестирование безопасности

В этой главе

- ✓ Сходство работы в областях тестирования и безопасности
- ✓ Обнаружение угроз безопасности с помощью моделирования
- ✓ Как использовать философию безопасности в различных видах тестирования

У кого-то идея тестирования безопасности может вызвать в воображении образы людей, осуществляющих высокотехнологичные и сложные атаки, которые обнаруживают немыслимые уязвимости в наших системах. Действительно, знание того, как работают системы, как их эксплуатировать и как использовать инструменты для обнаружения угроз, является ключевым условием успешного тестирования безопасности. Однако неверные представления о тестировании безопасности способствуют распространению идеи, что им занимаются только члены элитарного клуба, обладающие сверхчеловеческими техническими навыками. Однако тестирование безопасности — это не просто «взлом» систем, оно требует целенаправленного планирования и анализа для обнаружения угроз и определения их приоритетов. Все это включает в себя широкий спектр навыков и техник, часть из которых мы уже изучали в предыдущих главах.

Если мы посмотрим на алгоритмы работы специалистов в сфере безопасности, то обнаружим значительное совпадение с тем, что мы уже узнали на страницах этой книги. Специалисты по тестированию безопасности пытаются обнаружить и смягчить риски, которые могут повлиять на качество продуктов. Основное отличие заключается в том, что они фокусируются на характеристиках качества, связанных с безопасностью и конфиденциальностью. Осознав это сходство, мы можем начать путь к внедрению философии безопасности в наше тестирование. Как вы увидите далее, мы сможем использовать подходы к планированию и методы моделирования, изученные в предыдущих главах.

Прежде чем начать

В этой главе мы рассмотрим, как тесты безопасности могут быть интегрированы в другие виды тестирования. Поэтому предполагается, что вы уже прочитали главы 2, 4 и 5 или, по крайней мере, знакомы с рассмотренными в них концепциями. Кроме того, важно отметить, что здесь дается не исчерпывающее описание возможностей в области тестирования безопасности, а только описание общих методов.

11.1. РАБОТА С МОДЕЛЯМИ УГРОЗ

Лучший пример сходства между тестированием безопасности и уже изученными видами тестирования — моделирование угроз. В главе 2 мы узнали, как можно использовать модели для лучшего понимания работы наших систем и выявления рисков, которые могут повлиять на эти системы. Моделирование угроз имеет ту же основу. Модели угроз используются для того, чтобы понять, как работает система, и определить ключевые области, которые мы хотим защитить. Затем эти знания используются для определения потенциальных угроз, требующих устранения. Подобно моделированию и анализу рисков, которые мы проводили в предыдущих главах, ценность моделирования угроз заключается в том, что оно помогает определить и расставить приоритеты угроз. Создание модели угроз включает следующие шаги:

1) создание модели системы, которую мы можем использовать для анализа;
2) анализ модели на предмет потенциальных угроз высокого уровня;
3) глубокое изучение каждой угрозы с помощью техники, называемой «дерево атак».

Вспомните, что мы узнали о моделировании систем в главе 2: мы не обязательно должны моделировать всю систему за один раз. Разделение на небольшие фрагменты позволит легче воспринимать и выявлять потенциальные проблемы, которые могут остаться незамеченными при анализе системы в целом. Рассмотрим этот подход на примере `booking` API. Если более конкретно, то конечная точка `GET /booking/` должна быть доступна только для пользователей-администраторов.

11.1.1. Создание модели

Наш первый шаг в создании модели угроз — понять, что именно находится под угрозой. Когда мы рассматриваем вопрос о том, что хотим защитить, обычно имеем дело с конфиденциальными данными или процессами, связанными с этими данными. Поэтому, в отличие от моделей, которые мы создавали в предыдущих главах, нам требуется смоделировать поток и хранение данных. Вот почему многие модели угроз представлены в виде диаграмм потоков данных (data flow diagrams, DFD), иллюстрирующих поток информации: какие процессы используют данные и где они хранятся. DFD, созданная для конечной точки `GET /booking`, показана на рис. 11.1.

В примере модель DFD описывает процесс запроса администратором списка бронирований:

1. Администратор отправляет запрос на бронирование, предоставляя токен входа как часть запроса.
2. Токен извлекается из запроса в `booking` API и отправляется в `auth` API для проверки.
3. Если токен подтверждает, что администратор имеет доступ, отправляются данные бронирования. Если нет, то вместо этого возвращается код ошибки.

Модель указывает на факторы, которые целесообразно учесть при анализе системы на предмет потенциальных угроз. Используя стандартные обозначения DFD, мы можем обратить внимание на следующее:

- Внешний объект, взаимодействующий с процессом, должен быть администратором (на рисунке выделен прямоугольником).
- Выделенные текстом с верхним и нижним подчеркиванием ресурсы, токены сессий и бронирований требуется сохранить защищенными.
- Обозначенные кружками процессы обрабатывают входящие данные и наши активы.

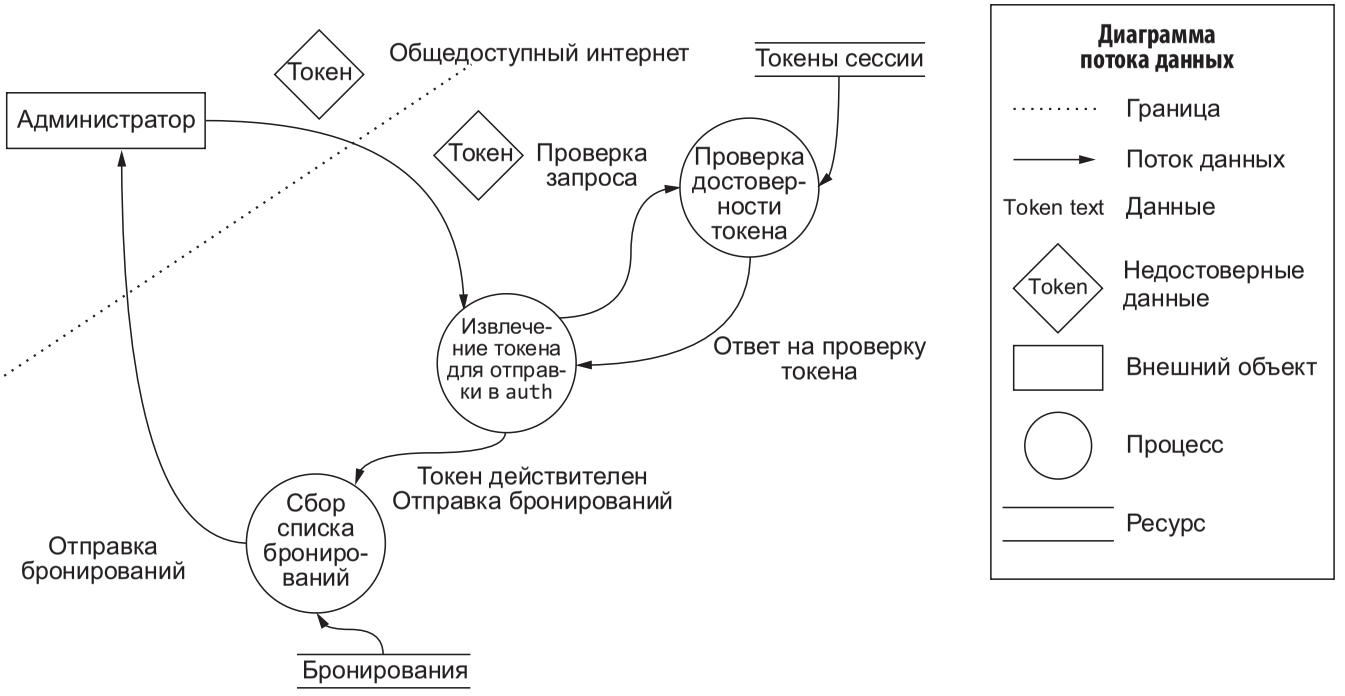

Рис. 11.1. Диаграмма потоков данных, показывающая, как обрабатываются данные для получения всех бронирований

- Имеются недостоверные данные, поступающие в систему в виде «токена» (обозначены ромбами).

Благодаря этим деталям мы можем разобраться в ожиданиях относительно обработки наших данных в разных ситуациях и взаимодействий с ними. Это дает основу для анализа нашей модели с целью обнаружения потенциальных угроз.

Сочетание моделей

Хотя использование подхода DFD является обычным при моделировании угроз, не бойтесь пробовать другие подходы, чтобы собрать важную для анализа информацию. DFD позволяет сосредоточиться на данных, но не отражает, какие механизмы были использованы для создания нашей системы и используемой инфраструктуры. Задача заключается в том, чтобы создать модель для анализа потенциальных угроз. Экспериментируйте, чтобы понять, как добавление других элементов в модель поможет вашему анализу.

Упражнение

Выберите небольшой фрагмент вашей системы и попытайтесь нарисовать DFD-модель его работы. Если необходимо, прибегните к помощи коллег, чтобы заполнить пробелы в модели или уточнить ваши предположения. Не бойтесь добавлять дополнительную информацию, которая может показаться вам полезной.

11.1.2. Обнаружение угроз с помощью STRIDE

Одной из причин, по которой тестирование безопасности считается достаточно сложной задачей, является многообразие способов, которыми безопасность нашей работы может быть поставлена под угрозу. Угрозы порождает широкий круг субъектов, от организованных групп преступников, желающих украсть данные для продажи, до активистов, наносящих ущерб сайтам по политическим мотивам (очень маловероятный сценарий для небольшой компании B&B). Разные субъекты используют различные векторы атак, которые мы должны учитывать при анализе систем. Именно поэтому, чтобы облегчить анализ угроз, были созданы такие инструменты, как STRIDE.

Как и эвристика, о которой мы узнали в главе 5, STRIDE связан с мнемоникой. Буквы в STRIDE обозначают различные аспекты безопасности, которые мы можем проанализировать с помощью модели DFD. Чтобы начать применять STRIDE, давайте разберемся, что означает каждая из букв этой аббревиатуры.

- *Спуфинг* (Spoofing) — это выдача себя за другого человека путем использования подложной информации. Например, злоумышленник может попытаться подделать данные для входа в систему. Или это может быть мошенник, выдающий себя за представителя доверенной организации, например банка, чтобы выведать у жертвы данные счета. Подделка может происходить разными способами, поэтому мы хотим быть уверены в возможности правильной аутентификации пользователя перед предоставлением ему доступа к конфиденциальной информации.
- *Фальсификация* (Tampering) подразумевает получение злоумышленниками возможности изменять код или данные в системе или при их передаче из одного места в другое. Одним из распространенных примеров фальсификации является «атака посредника», когда HTTP-запросы, отправленные по сети, перехватываются и изменяются злоумышленником до того, как придут в пункт назначения. Другой пример — внедрение вредоносных программ, таких как вирусы, в сети или на ПК, чтобы нарушить работу систем. Чтобы защитить себя от этих типов атак, необходимо обеспечить безопасность обмена данными между системами и проверку подлинности поступающих или отправляемых данных.
- *Отрицание* (Repudiation). Если спуфинг и фальсификация связаны со способами использования системы, то отрицание касается возможности злоумышленника утверждать, что он не совершал атаки. Например, если мы не регистрируем детали взаимодействия объектов с нашими системами, то не можем определить, подвергаются ли они атакам. Отсутствие информации может быть использовано злоумышленниками, чтобы продолжать атаки, оставаясь незамеченными, и заявить о своей невиновности, если впоследствии их обвинят в злонамеренном поведении. Поэтому мы стремимся обеспечить не только возможность отслеживать подозрительное поведение, но и гарантировать, что мониторинг не может быть обойден или фальсифицирован во время атаки.
- *Раскрытие информации* (Information disclosure) связано с утечкой сведений лицам, не имеющим права доступа к ним. Существует множество примеров утечки различных типов конфиденциальной или защищенной информации, но реже обсуждается вопрос о том, каким образом может произойти такая утечка. Она может быть связана со слабым контролем авторизации, неправильно настроенным контролем доступа либо с тем, что конфиденциальная информация не защищена от таких атак, как спуфинг или фальсификация.
- *Отказ в обслуживании* (Denial of service, DoS) имеет своей целью помешать доступу к информации или услугам. Вместо того чтобы пытаться проникнуть

в защищенные зоны, DoS-атаки стремятся вывести системы из строя или сделать их недоступными, чтобы нарушить работу организации или получить выкуп. Например, злоумышленники могут осуществлять распределенную атаку, перегружая сервисы большим количеством запросов, или инструменты выкупного ПО (ransomware) могут блокировать доступ отдельных лиц или организаций к своим сервисам до тех пор, пока не будет выплачен выкуп.

- *Повышение привилегий* (Elevation of privilege) относится к атакам, которые позволяют злоумышленнику повысить свой уровень привилегий. Такие атаки используют уязвимости и эксплойты для получения доступа. В качестве примера можно привести злоумышленников, которые находят способы получить доступ к учетным записям других пользователей (горизонтальное повышение) или обходят контроль доступа, чтобы повысить свои привилегии до уровня администратора (вертикальное повышение).

Изучение элементов STRIDE помогает разобраться не только с различными типами атак, но и с тем, как они могут сочетаться друг с другом. Например, спуфинг может быть использован для получения доступа, чтобы злоумышленник мог осуществить атаку типа «раскрытие информации». Это затрудняет тестирование безопасности, поскольку уязвимость в одной части системы может оказать значительное влияние на другую. Поэтому модель STRIDE должна сочетаться с моделью DFD, позволяя определять, как различные части системы могут быть уязвимы для различных типов атак.

Например, мы можем начать со спуфинга. Такой атаке может подвергаться администратор. Вместо легитимного пользователя отправителем токена может быть злоумышленник. Поэтому пометим эту часть нашей DFD-модели как потенциально уязвимую к атаке спуфинга (рис. 11.2).

На диаграмме мы использовали наши знания о системе и понимание STRIDE, чтобы определить уязвимости для атак разных типов. Помимо спуфинга, обозначены следующие угрозы:

- `auth` API и `booking` API могут быть подвержены DoS-атакам. `auth` API особенно важен, потому что если он перестанет работать, большая часть системы станет недоступной;
- если злоумышленник подделает или подменит токены, атака может остаться незамеченной из-за отсутствия логирования и мониторинга в `auth` API;
- список бронирований может стать мишенью для раскрытия информации. Отсутствие шифрования брони означает, что похищенные данные можно будет легко использовать.

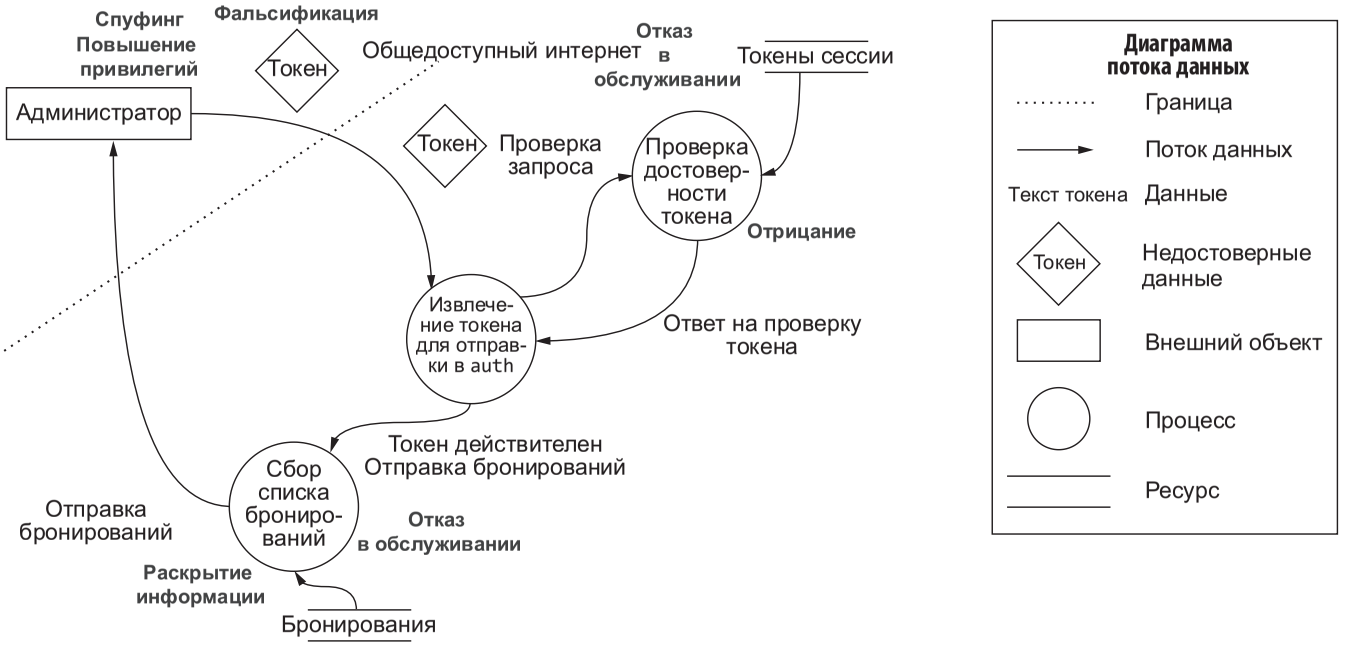

Рис. 11.2. Диаграмма потоков данных, обновленная для отображения потенциальных угроз с помощью STRIDE

Хотя мы можем не знать конкретных деталей того, как будет производиться атака, мы уже начали анализировать потенциальные уязвимости. Это дает возможность начать определять приоритеты. Например, нас могут больше волновать DoS-атаки, чем раскрытие информации. Если же мы хотим определить приоритеты конкретных угроз для наших систем, нужно разделить элементы STRIDE на более мелкие. Для этого служат деревья угроз.

Упражнение

Используя созданную вами DFD-модель системы, попытайтесь применить к ней STRIDE. Укажите в каждой из областей модели типы атак, к которым она может быть уязвима. Обсудите с коллегами способы их предотвращения, которые уже реализованы или планируются в будущем.

11.1.3. Создание деревьев угроз

Деревья угроз, или деревья атак, позволяют разложить общие угрозы, выявленные с помощью STRIDE, на частные угрозы реальных атак, которые могут быть осуществлены против системы. В качестве примера вернемся к угрозе спуфинга. Мы можем начать разбирать множество различных способов, которыми злоумышленник может подделать идентификаторы реального пользователя-администратора. Начнем с высокоуровневых способов спуфинга, а затем разделим их на более конкретные атаки (рис. 11.3).

Дерево угроз состоит из нескольких уровней. Разные виды атак детализируются при переходе от абстрактной идеи к конкретным техникам. Используя древовидную структуру, мы имеем возможность охватить широкий круг вопросов, например, перечислить потенциальные высокоуровневые спуфинг-атаки, такие как угадывание, кража учетных данных или социальная инженерия. Выбрав одну тему, можно раскрыть ее, чтобы выявить атаки, к которым может быть уязвима система. Например, утечка токенов доступа возможна в результате собственных ошибок или через сторонние инструменты, от которых мы зависим.

Дерево угроз полезно, поскольку позволяет обозначить большое разнообразие угроз. Это продемонстрировано в другом примере, сфокусированном на DoS-атаках (рис. 11.4).

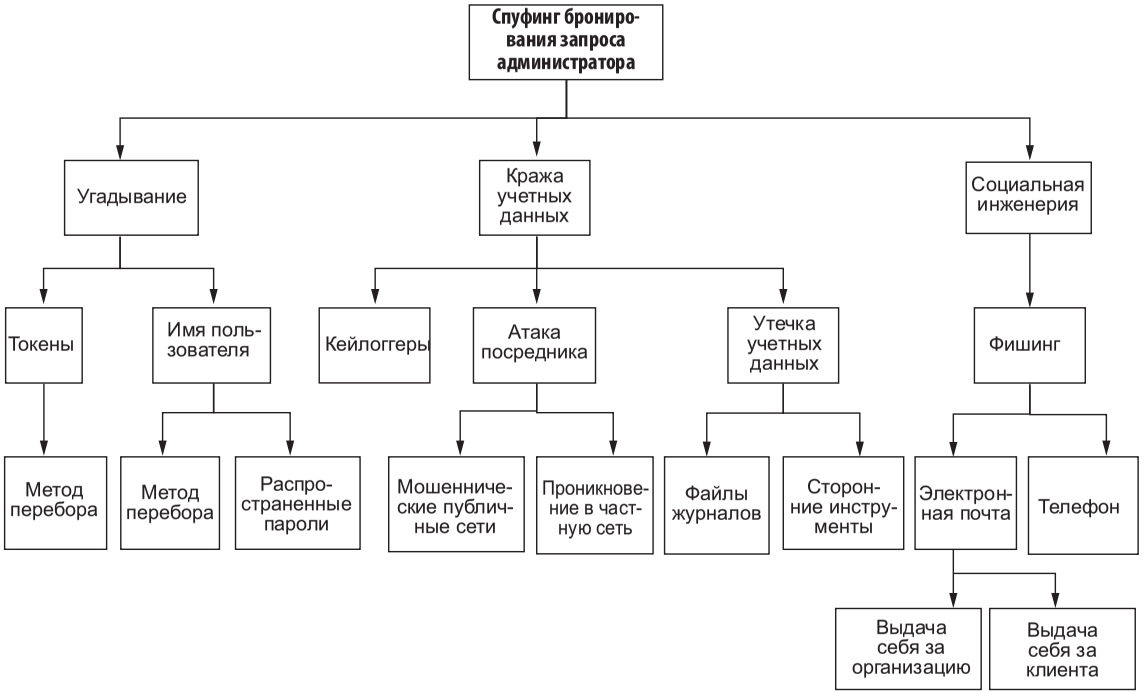

Рис. 11.3. Дерево угроз, раскрывающее виды спуфинг-атак против нашего приложения

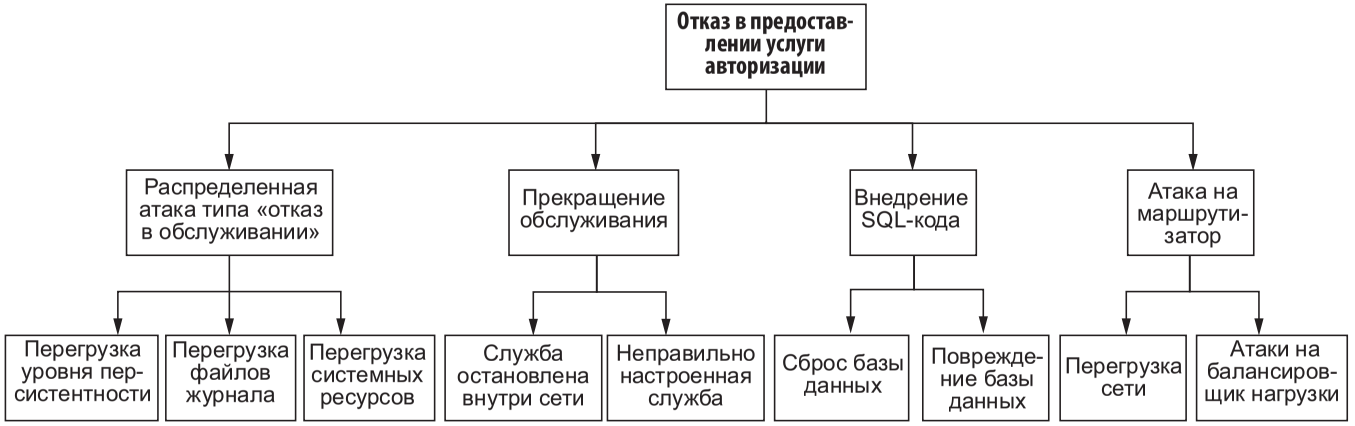

Рис. 11.4. Дерево угроз, раскрывающее виды DoS-атак против нашего приложения

Мы видим разнообразный спектр DoS-атак, к которым может быть уязвим наш продукт. Дерево отражает наиболее известные типы атак, например распределенную DoS, когда служба подвергается бомбардировке запросами из самых разных мест. Но оно также демонстрирует DoS-атаки, которые могут использовать неправильную конфигурацию наших служб, например случайное размещение в открытом доступе функций, позволяющих отключать сервисы (это возможно в случае с API-фреймворком Spring-boot-actuator, который использует наша API-песочница).

Стоит отметить, что приведенные примеры деревьев ни в коем случае не содержат исчерпывающий список всех атак, к которым мы можем быть уязвимы. Содержание дерева угроз зависит от того, что мы считаем наиболее целесообразным для исследования, то есть от нашего понимания системы и знаний о типах угроз. При создании дерева угроз целесообразно учесть следующее:

- Создавайте деревья угроз совместно с коллегами. Различные точки зрения и опыт дают разнообразные идеи о том, к чему мы можем быть уязвимы.
- Изучите примеры прошлых атак и материалы, которыми поделились исследователи безопасности по конкретным темам или типам атак.
- Изучите известные уязвимости в инструментах и технологиях, которые вы используете, с учетом миграции этих уязвимостей.

Дерево угроз позволяет нам дать волю воображению. Чем больше мы можем добавить в дерево угроз, тем выше вероятность обнаружения и смягчения уязвимостей. Если постараться рассмотреть как можно большее количество потенциальных атак, то следующий шаг по определению приоритетов и смягчению угроз будет более эффективным.

Упражнение

Выберите один из элементов STRIDE в вашей модели и попытайтесь создать дерево угроз, отражающее различные способы, которыми может быть атакована ваша система. Изучите различные способы атаки и добавьте их в свое дерево.

11.1.4. Минимизация угроз

В конечном счете, обсуждая угрозы в контексте тестирования безопасности, мы работаем с определенным типом риска. Поэтому после завершения моделирования и обнаружения угроз/рисков безопасности следующим шагом будет

определение приоритетности рисков. На первом этапе мы можем использовать подход, который изучили в главе 3: определим приоритеты рисков путем рассмотрения вероятности и степени серьезности каждой угрозы и установим, на чем следует сосредоточиться. Если риск безопасности имеет высокую вероятность реализации и нанесения серьезного ущерба нашим системам и бизнесу, то мы должны стремиться к его снижению.

Проект Open Web Application Security Project (OWASP https://owasp.org) продвигает методику определения приоритетов DREAD, созданную компанией Microsoft (которая также предлагает концепцию моделирования угроз и необходимые наборы инструментов: http://mng.bz/deND). Составляющие DREAD:

- *Ущерб* (Damage). Каков будет ущерб для нас и наших пользователей?
- *Воспроизводимость* (Reproducibility). Насколько легко злоумышленники могут воспроизвести угрозу?
- *Возможность выполнения* (Exploitability). Насколько легко использовать уязвимость?
- *Затронутые пользователи* (Affected users). Сколько пользователей будет затронуто?
- *Возможность обнаружения* (Discoverability). Насколько легко злоумышленнику обнаружить уязвимость?

В отличие от более распространенного подхода к определению приоритетов на основе вероятности и серьезности, DREAD предлагает лучшую детализацию, поскольку каждый пункт оценивается по десятибалльной шкале. Рассмотрим пример: мы опасаемся, что злоумышленник может использовать внедрение SQL-кода, в результате чего будет раскрыта конфиденциальная информация, что позволит закрыть приложение и вызвать отказ в обслуживании. Применив методику DREAD, мы получим следующие оценки:

- *Ущерб: 7*. Конфиденциальные данные, которые мы хотим скрыть, могут быть раскрыты.
- *Воспроизводимость: 8*. После обнаружения атака может быть легко повторена.
- *Возможность выполнения: 3*. Реализация атаки требует глубоких знаний о внедрении SQL-кода и структуре наших баз данных.
- *Затронутые пользователи: 10*. Утечка информации из баз данных может повлиять на всех наших пользователей.

- *Возможность обнаружения: 8*. Наши API могут подвергаться различным атакам с использованием внедрения кода.

Рассчитаем среднюю оценку путем деления суммы баллов на пять. В нашем примере оценка приоритета будет равна (7+8+3+10+8)/5=7,2. Применим эту модель к каждой угрозе и расположим угрозы по значениям приоритетов — от бóльших к меньшим.

Моделирование угроз показывает, что тестирование безопасности не требует сакральных знаний, как обычно думают люди. В моделировании угроз используются подходы, многие из которых мы освоили в других видах тестирования, в частности, связанные с анализом рисков. Оттачивая эти навыки применительно к моделированию угроз, мы можем выявлять различные риски, не вникая в технические детали атак, но помогая нашей команде создавать более безопасные приложения.

11.2. ИСПОЛЬЗОВАНИЕ ФИЛОСОФИИ БЕЗОПАСНОСТИ ПРИ ТЕСТИРОВАНИИ

Моделирование угроз создает предпосылки для повышения безопасности, но мы можем пойти дальше, включив философию безопасности в другие виды тестирования. Для этого надо использовать результаты моделирования угроз при планировании других видов тестирования.

11.2.1. Тестирование безопасности на этапе проектирования API

Модели угроз предпочтительно создавать совместно с другими участниками команды. Мы уже отмечали, что разнообразие идей и опыта помогает в моделировании угроз. Кроме того, совместная работа обеспечивает информирование команды об угрозах и, что очень важно, позволяет согласовать план по их устранению.

Как мы узнали в главе 4, предварительное обсуждение дизайна API в команде дает следующие возможности:

- узнать подробнее о целях создания функций;
- убедиться, что все четко понимают, что нам нужно создать;

- зафиксировать потенциальные риски, которые могут угрожать качеству продукта.

Совместное моделирование угроз в процессе проектирования API помогает в реализации последнего пункта. Когда угрозы определены, мы можем обсудить и решить, как их уменьшить. Это может привести к изменениям дизайна. Например, обсуждение техники сброса паролей может выявить необходимость предотвращения угрозы намеренного сброса пароля пользователя с блокировкой его учетной записи. Поэтому мы можем расширить соответствующую функциональность, добавив отправку письма с подтверждением сброса пароля и ограничения по времени для сброса. Это помогает команде, поскольку проще повысить безопасность на этапе создания функции, чем дорабатывать ее позже (возможно, после того, как атака уже произошла).

Естественно, не стоит создавать модель угроз для каждого изменения. Мы как команда должны определять, когда моделирование угроз необходимо. Но если мы будем проводить моделирование угроз регулярно, то, совершенствуя имеющиеся модели для экономии времени, сможем выработать привычку встраивать безопасность в дизайн еще до начала работы над кодом, что сэкономит много времени и исключит головную боль на более поздних этапах работы.

11.2.2. Исследовательское тестирование безопасности

До сих пор наши исследования рисков безопасности носили несколько абстрактный характер, концентрируясь на том, что может произойти. А как насчет исследования продукта для обнаружения реальных рисков, которые уже существуют в системе? Здесь мы можем использовать навыки исследовательского тестирования, полученные в главе 5. Применим тот же подход, что и при тестировании в течение определенного времени, но с упором на безопасность.

Составление устава для тестирования безопасности

В главе 5 мы узнали об уставах для руководства тестированием, основанных на рисках, по которым мы считаем важным получить больше информации. Когда речь идет об угрозах безопасности, уставы составляются точно так же.

И здесь снова будет полезно смоделировать угрозы для определения уставов в отношении угроз безопасности. Хотя некоторые угрозы могут быть устранены на ранней стадии путем изменения дизайна API, все равно целесообразно

исследовать продукт на наличие угроз (особенно если мы выполняем тестирование безопасности на поздней стадии проекта). Поэтому нужно выбрать конкретные угрозы из дерева угроз, превратить их в уставы и приступить к работе.

Здесь стоит упомянуть еще один источник, из которого можно черпать вдохновение, — OWASP Top 10 (https://owasp.org/Top10/). Этот список десяти наиболее распространенных угроз для веб-приложений можно использовать для создания уставов, ориентированных на безопасность. Его применение полезно сочетать с моделированием угроз, поскольку OWASP Top 10 предлагает другой взгляд на потенциальные угрозы. Моделирование угроз хорошо лишь настолько, насколько хороши люди, которые его выполняют, а это означает, что в анализе могут быть пробелы. Обращаясь к Top 10, мы можем убедиться, что пробелы, которые неизбежно присутствуют в нашей работе, не являются наиболее распространенными угрозами.

OWASP Top 10 составлен для различных типов веб-приложений, поэтому не все пункты будут применимы в контексте безопасности API. Например, межсайтовый скриптинг неактуален, если нет фронтэнда для визуализации. Тем не менее этот список полезен для создания устава исследовательского тестирования. Чтобы продемонстрировать это, давайте создадим устав на основе седьмого пункта списка «Сбои идентификации и аутентификации». Запишем его в формате устава следующим образом:

> Изучите `auth` API...
>
> с помощью Burp Suite и списка распространенных паролей,
>
> чтобы узнать, можно ли взломать пароль методом перебора.

Рассмотрим, как подобный устав может быть выполнен с помощью инструмента Burp Suite, чтобы узнать больше о потенциальной угрозе в `auth` API.

Исследование с использованием инструментов

Задача устава — выяснить, как реагирует `auth` API на перебор паролей. Будет ли он только принимать запросы? Начнет ли он ошибаться через некоторое время? Сможем ли мы войти в систему? Для этого воспользуемся инструментом тестирования безопасности Burp Suite, который содержит набор средств для сканирования и атак. Мы будем использовать Burp Suite Community Edition (https://portswigger.net/burp/communitydownload), в котором имеется все необходимое для проведения атаки методом перебора.

Инструменты защиты

Следует отметить, что профессиональная платная версия Burp Suite имеет ряд дополнительных инструментов для обнаружения уязвимостей, которые можно использовать в сеансах исследовательского тестирования. Но стоит она недешево. Если вас беспокоит цена, то можете использовать другие инструменты, такие как OWASP ZAP (https://www.zaproxy.org/), который является бесплатным и имеет открытый исходный код, но научиться пользоваться им гораздо сложнее.

Чтобы настроить сеанс тестирования, нужно выполнить следующие шаги:

1. Запустим нашу тестовую песочницу API. Можно запустить все приложение, используя скрипты запуска, либо, поскольку сейчас нас интересует только `auth` API, можно загрузить только его.
2. Установим и откроем Burp Suite Community Edition. В версии Community Edition мы можем выбрать только временный проект для запуска, что приемлемо. Выберем настройки Burp Suite по умолчанию.

Настройка прокси-сервера Burp Suite

Одной из функций Burp Suite является прокси-сервер, который используется для перехвата запросов и ответов для дальнейшего анализа. Хотя мы не будем использовать его в этой демонстрации, нужно знать, что он работает на порте 8080 — том же, который использует API песочницы. Поэтому если вы столкнетесь с проблемой при загрузке API песочницы, получая вместо этого ответ от Burp Suite, перейдите в Proxy > Options в Burp Suite и либо отключите прокси в разделе Proxy Listeners, либо нажмите на запись в таблице, выберите Edit, а затем измените поле Bind to port на 8081 или любое другое подходящее значение. Затем нужно перезапустить API песочницы.

Когда все готово, мы можем приступить к созданию атаки. Для этого воспользуемся функцией Intruder в Burp Suite, которую можно найти в списке вкладок в верхней части окна приложения. Прежде чем мы сможем указать детали для атаки Intruder, потребуется HTTP-запрос для отправки с нашими данными. Поскольку в центре нашего внимания находится конечная точка `auth` API `POST /auth/login`, нужно извлечь детали HTTP-запроса, однако сейчас это не важно, поэтому можно использовать следующий HTTP-запрос:

```
POST /auth/login HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
    "username": "admin",
    "password": "fuzzme"
}
```

Теперь мы можем приступить к настройке атаки Intruder, начиная со вкладки Target. На вкладке Target введем узел и порт приложения, которое мы собираемся взломать. Если работает полная версия песочницы, то устанавливаем следующие параметры:

- Host: localhost
- Port: 8080

Если работает только `auth` API, то настройки такие:

- Host: localhost
- Port: 3004

Это должно совпадать с заголовком Host из нашего HTTP-запроса. В противном случае мы получим ошибку.

Далее откроем вкладку Positions, где есть поле для вставки нашего HTTP-запроса `POST /auth/login`. Теперь нужно настроить его, указав, какие поля мы хотим проверить. Поскольку наша цель — попробовать разные пароли, мы должны выбрать значение `fuzzme` в JSON-объекте (без кавычек), а затем нажать кнопку Add, которая находится в правой части окна приложения. Это сообщит Burp Suite, что мы намерены заменить содержимое `fuzzme` (дословно — «подставь меня») различными вариантами перебора, которые мы установили на вкладке Payloads.

После того как запрос настроен, нужно добавить список атак, которые мы хотим осуществить. Для этого на вкладке Payloads используем следующие настройки:

- *Payload set* (набор полезных нагрузок) — это значение остается равным `1`, так как мы задали только один параметр для изменения в нашем HTTP-запросе: `fuzzme`.
- *Payload type* (тип полезной нагрузки). Burp Suite предлагает использовать разные типы полезной нагрузки в зависимости от вида атаки. Поскольку

мы будем перебирать список распространенных паролей, нам подойдет тип `Simple list` (простой список).

- *Payload options* (параметры полезной нагрузки) — здесь мы добавляем список распространенных паролей, которые хотим использовать. Чтобы получить список распространенных паролей, можно выполнить поиск в интернете по ключевым словам *Common Passwords txt list* (я нашел такой список на http://mng.bz/rnYg). Скопируйте список и нажмите для его добавления Paste в Payload Options.

Итак, мы отправили Burp Suite на нужный узел, обновили HTTP-запрос с заданным полем пароля и добавили список паролей, которые нужно проверить. Нажмите кнопку Start attack в правом углу окна, чтобы начать сеанс. На экран будут выведены результаты, похожие на представленные на рис. 11.5.

Results | Target | Positions | Payloads | Resource Pool | Options

Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length
0		200	☐	☐	138
1	123456	403	☐	☐	101
2	12345	403	☐	☐	101
3	123456789	403	☐	☐	101
4	password	200	☐	☐	138
5	iloveyou	403	☐	☐	101
6	princess	403	☐	☐	101
7	1234567	403	☐	☐	101
8	rockyou	403	☐	☐	101
9	12345678	403	☐	☐	101
10	abc123	403	☐	☐	101
11	nicole	403	☐	☐	101
12	daniel	403	☐	☐	101
13	babygirl	403	☐	☐	101
14	monkey	403	☐	☐	101
15	lovely	403	☐	☐	101
16	jessica	403	☐	☐	101
17	654321	403	☐	☐	101
18	michael	403	☐	☐	101
19	ashley	403	☐	☐	101
20	qwerty	403	☐	☐	101
21	111111	403	☐	☐	101
22	iloveu	403	☐	☐	101
23	000000	403	☐	☐	101
24	michelle	403	☐	☐	101

Рис. 11.5. Пример отчета о вторжении, выполненного в Burp Suite

Мы видим в результатах подробную информацию о том, какой пароль отправляется, а также код состояния ответа. Можно щелкнуть на каждом запросе, чтобы получить более подробную информацию. Что оказалось интересным

в этой сессии, так это количество запросов и их коды состояния. В частности, мы смогли получить код 200 с одним из паролей, а все остальные вернули код 403, а не ошибку, связанную с тем, что запросы отклоняются системой. Это говорит о том, что у нас есть следующие проблемы безопасности:

- пароль по умолчанию для администратора очень слабый, это четвертый по распространенности пароль в списке, который мы использовали;
- отсутствует защита от перебора паролей.

Таким образом, краткий сеанс исследовательского тестирования выявил две потенциальные проблемы, требующие решения. Если сделать шаг назад и посмотреть, что происходило во время сеанса, то он ничем не отличается от других исследовательских сеансов, которые мы анализировали в этой книге. Мы следовали уставу, чтобы узнать о конкретном риске, и использовали инструментарий для поддержки нашего исследования. Просто в центре внимания находился другой тип риска. Так мы еще раз продемонстрировали, что сочетание философии безопасности и существующих видов тестирования может быстро принести плоды.

Упражнение

Попробуйте провести сеанс исследовательского тестирования с помощью функции Intruder в Burp Suite. Области приложения, которые можно рассмотреть для проверки:

- `POST /message` — проверьте полезную нагрузку сообщения на наличие внедрений SQL;
- `GET /booking/` — проверьте строку запроса `roomid` на наличие потенциальных уязвимостей;
- `POST /auth/` — загрузите в полезную нагрузку для входа содержимое, которое может заставить API отправить обратно ошибку сервера.

11.2.3. Автоматизация и тестирование безопасности

Пример сеанса исследовательского тестирования продемонстрировал, что с помощью инструментов безопасности мы можем узнать больше об угрозах. При этом мы можем применять и другие инструменты, например, чтобы создать автоматизированный пайплайн, требующий минимальных операций. В главе 6 мы узнали, как автоматизированные проверки могут использоваться в качестве индикаторов, предупреждающих об изменениях в системе. Благодаря этому мы можем проанализировать, повлияло ли это изменение на качество приложения.

Аналогичные решения возможны в контексте безопасности с помощью инструментов, сообщающих, произошли ли изменения, которые затрагивают возможные уязвимости.

Когда мы создаем системы, то полагаемся на широкий спектр сторонних библиотек и зависимостей, а также используем в коде популярные модели проектирования. Хотя мы стараемся, чтобы в продукте не было уязвимостей, предотвратить все возможные угрозы невозможно. Позже может обнаружиться, что шаблоны проектирования уязвимы для определенных типов атак, а зависимости также могут иметь уязвимости (например, пока я писал эту главу, стало известно об уязвимости log4js). Главное, как мы реагируем на обнаружение этих уязвимостей. Если они известны злоумышленникам, то создание использующих их инструментов — лишь вопрос времени. Для того чтобы защитить себя, мы должны вовремя узнать о новой уязвимости, определить, существует ли она в нашей системе, и устранить до того, как она станет серьезной проблемой.

Чтобы выявлять существующие или новые уязвимости в кодовой базе, мы можем использовать автоматизацию. Давайте рассмотрим два способа применения инструментов для выявления потенциальных уязвимостей.

Статический анализ

Анализ нашей кодовой базы на наличие проблем является отличным способом обнаружения потенциальных угроз. Такие инструменты, как SonarQube и линтеры с открытым исходным кодом, имеют ряд плагинов или интеграций, проверяющих код до его развертывания и предупреждающих об угрозах. На самом деле их так много, что выбор может ошеломить (http://mng.bz/Vy7X).

Рассмотрим пример статического анализа, который можно использовать как часть пайплайна: проанализируем кодовую базу `auth` API с помощью инструмента SpotBugs (https://github.com/spotbugs/spotbugs) и плагина Find Security Bugs (https://find-sec-bugs.github.io/). Сначала мы добавим следующий плагин в `auth` API в файл pom.xml:

```
<plugin>
    <groupId>com.github.spotbugs</groupId>
    <artifactId>spotbugs-maven-plugin</artifactId>
    <version>4.5.0.0</version>
    <configuration>
        <plugins>
            <plugin>
                <groupId>com.h3xstream.findsecbugs</groupId>
                <artifactId>findsecbugs-plugin</artifactId>
                <version>1.10.1</version>
```

```
            </plugin>
        </plugins>
    </configuration>
</plugin>
```

Откроем терминал и выполним команду `mvn spotbugs:check`, чтобы запустить анализ нашей кодовой базы. По его итогам будет выдан список предупреждений. Например, в какой-то момент было обнаружено следующее:

```
[ERROR] Medium: Cookie without the HttpOnly flag could be read by a malicious
    script in the browser [com.automationintesting.api.AuthController] At
    AuthController.java:[line 30] HTTPONLY_COOKIE
[ERROR] Medium: Cookie without the secure flag could be sent in clear text if
    a HTTP URL is visited [com.automationintesting.api.AuthController] At
    AuthController.java:[line 30] INSECURE_COOKIE
```

Таким образом, мы можем обновлять наш код, чтобы делать его более безопасным.

Проверка зависимостей

Статический анализ полезен для анализа нашего кода на предмет потенциальных уязвимостей, но как насчет библиотек, от которых мы зависим при создании систем? Обеспечение актуальности зависимостей является важной частью защиты наших систем, причем их проверка стала стандартом на таких сайтах, как GitHub. Последний регулярно сканирует кодовые базы, информируя об обнаруженных уязвимостях в наших зависимостях.

Однако проблемы обнаруживаются уже после фиксации изменений нашего кода и его размещения на стороннем сайте, что не всегда возможно. В качестве альтернативного подхода мы можем использовать такие инструменты, как OWASP Dependency-Check (https://jeremylong.github.io/DependencyCheck/), который позволяет выполнять проверки локально до фиксации кода.

Давайте рассмотрим, как мы можем использовать проверку зависимостей в `auth` API для обнаружения уязвимостей. Для начала добавьте плагин Dependency-Check в `auth` API в файл pom.xml:

```
<plugin>
    <groupId>org.owasp</groupId>
    <artifactId>dependency-check-maven</artifactId>
    <version>6.5.0</version>
    <executions>
        <execution>
            <goals>
                <goal>check</goal>
```

```
            </goals>
        </execution>
    </executions>
</plugin>
```

Откройте терминал и выполните команду `mvn verify`. После выполнения всех автоматических проверок мы увидим, что Dependency-Check запустился и создал отчет, который сохраняется в файле target/dependency-check-report.html. Этот отчет содержит перечень зависимостей, на которые мы полагаемся, а также указания, какие из них имеют известные уязвимости. Например, быстрая проверка `auth` API показала, что зависимость `jackson-databind` имеет критическую уязвимость, которую нужно немедленно устранить, обновив зависимость до новой версии.

Опора на автоматизацию

Мы рассмотрели всего лишь два примера инструментов, которые можно добавить в пайплайны для сканирования кодовой базы и информирования нас о потенциальных рисках безопасности. Но существует значительно большее число инструментов, которыми можно воспользоваться. Например, уже знакомый нам по сеансу исследовательского тестирования Burp Suite имеет функции командной строки (некоторые из них платные), которые можно использовать как часть пайплайна (OWASP ZAP proxy имеет аналогичные функции). Такие инструменты не требуют больших инвестиций, если мы рассматриваем случай, когда автоматизация помогает повышению безопасности.

Однако следует предупредить: подобно тому как автоматизированные регрессионные проверки не обнаруживают все проблемы, инструменты сканирования безопасности не смогут найти все уязвимости. Они помогут улучшить наше тестирование безопасности, но автоматизация не должна создавать ложное ощущение безопасности (извините за каламбур), в результате чего мы можем столкнуться с неприятными сюрпризами от злоумышленников.

11.3. ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ КАК ЧАСТЬ СТРАТЕГИИ

Безопасность — обширная тема, и к сожалению, существует широкий спектр способов, с помощью которых находятся уязвимости для атак. Однако эта глава показала, что многие навыки и подходы, используемые в тестировании безопасности, ничем не отличаются от изученных в предыдущих главах. Мы можем рассматривать тестирование безопасности как философию, которая использует различные методы тестирования, такие как моделирование, анализ,

автоматизация и исследовательское тестирование с акцентом на проблемы безопасности, но в рамках более широкой стратегии. Таким образом, мы можем вписать тестирование безопасности в модель стратегии (рис. 11.6).

Рис. 11.6. Модель, показывающая, что принципы безопасности приложений необходимо учитывать во всей стратегии тестирования

Мы увидели, как моделирование угроз может проводиться одновременно с тестированием дизайна API, что позволяет стимулировать членов команды учитывать и эти проблемы в своей работе. Мы узнали, что обнаружение потенциальных угроз в наших системах по своей природе является исследовательским тестированием. Это позволяет организовывать сеансы тестирования, направленные на конкретные угрозы безопасности. И наконец, мы знаем, что можем использовать автоматизированные инструменты, предупреждающие о проблемах безопасности, чтобы эффективно поддерживать системы и защищать их от угроз. Все эти подходы ничем не отличаются от того, что многие команды уже делают, стремясь создавать качественные продукты. Просто добавив в тестирование философию, ориентированную на безопасность, мы можем внести существенный вклад в создание более безопасных систем.

ИТОГИ

- Тестирование безопасности схоже с другими видами тестирования.
- В нем используются общие для разных техник тестирования методы и навыки, такие как моделирование и анализ рисков, связанных с характеристиками качества.

- Мы можем использовать моделирование угроз в наших системах для обнаружения и определения приоритетов тех угроз, которые хотим ослабить.
- Моделирование угроз начинается с создания модели системы с помощью таких инструментов, как диаграммы потоков данных.
- Для выявления угроз целесообразно применить к модели инструмент STRIDE. Аббревиатура STRIDE означает спуфинг, фальсификацию, отрицание, раскрытие информации, отказ в обслуживании и повышение привилегий.
- Мы можем определить конкретные типы атак, создавая деревья угроз на основе каждого из элементов STRIDE.
- После идентификации атак рекомендуется определить приоритеты угроз, используя технику DREAD.
- Мы можем использовать результаты моделирования угроз в планировании других мероприятий по тестированию.
- Моделирование угроз при тестировании дизайна API позволяет выявлять угрозы на стадии проектирования и изменять наши планы для их смягчения.
- Можно использовать исследовательское тестирование, создавая и выполняя уставы, которые сфокусированы на угрозах безопасности.
- В сеансах исследовательского тестирования целесообразно применять инструменты тестирования безопасности, такие как Burp Suite и ZAP.
- Сканеры и линтеры безопасности, такие как SpotBugs или Dependency-Check, позволяют обнаруживать уязвимости в нашем коде или используемых библиотеках.
- Философия тестирования безопасности может применяться в рамках модели стратегии тестирования.

12 Тестирование в продакшене

В этой главе

- ✓ Что такое тестирование в продакшене
- ✓ Важность тестирования в продакшене как части стратегии тестирования
- ✓ Как определить, что именно отслеживать в продакшене
- ✓ Настройка инструментов для проведения тестирования в продакшене
- ✓ Альтернативные способы использования тестирования в продакшене

Когда в 2011 году Эрик Рис (Eric Ries) написал книгу «The Lean Startup»[1], предложенные им стратегия и философия оказали влияние на различные отрасли, включая разработку программного обеспечения. В центре внимания Эрика лежит важность качества и затрат. В своем блоге на linkedin.com (http://mng.bz/xMO8) он написал:

[1] Рис Э. «Бизнес с нуля».

Для коммерческой компании качество определяется тем, чего хочет клиент. Поэтому если мы не ориентируемся на его желания, то время на тщательную доработку деталей на самом деле тратится впустую, потому что в итоге мы отдаляем продукт от того, чего хочет клиент.

Из отношения Эрика к качеству можно извлечь два урока. Первый заключается в том, что качество «определяется тем, что хочет клиент», и эту тему мы подробно обсуждали выше. Второй состоит в том, что требуются подходы, которые позволят оценивать поведение наших пользователей и систем, чтобы получить информацию о качестве нашего продукта. Возможности здесь весьма обширны, например, в области тестирования следует обратить внимание на подход, основанный на тестировании в процессе производства.

Фраза «тестирование в продакшене» может вызвать чувство тревоги. В голову приходят мысли о появлении случайных тестовых данных на реальных сайтах, сбое производственных систем или небезопасных учетных записях, имеющих уязвимости для злоумышленников. Однако, размышляя о тестировании в продакшене, важно сделать шаг назад и напомнить себе о главной цели тестирования.

Как мы уже говорили ранее, тестирование — это изучение представления о нашем продукте (того, что мы хотим) и его реализации (того, что мы имеем), чтобы привести их в соответствие друг с другом. Для достижения этой цели мы можем испытывать наши продукты различными способами и наблюдать за тем, что с ними происходит. Но мы также можем многому научиться, наблюдая за тем, как их используют другие. Поэтому когда мы говорим о тестировании в продакшене, речь идет не столько об обнаружении проблем, сколько о сборе и анализе информации о том, как пользователи взаимодействуют с нашей системой. Это станет более понятным, когда мы рассмотрим, как начать тестирование в продакшене.

12.1. ПЛАНИРОВАНИЕ ТЕСТИРОВАНИЯ В ПРОДАКШЕНЕ

Тестирование в продакшене обусловлено желанием узнать, как наши пользователи взаимодействуют с нашим продуктом и как наша система ведет себя в «живом» окружении. Соответственно, нам требуется план по сбору показателей и их анализу. Поэтому, прежде чем приступить к внедрению инструментов для тестирования в продакшене, необходимо получить ответы на вопрос: какие

показатели мы должны отслеживать и как определить, указывают ли их значения на снижение качества?

12.1.1. Что отслеживать

Чтобы определить, что нужно отслеживать, можно использовать методы, разработанные специалистами по надежности сайтов (SRE-инженерами). Зададим два вопроса:

- Каков необходимый уровень доступности и надежности для каждой характеристики продукта?
- Какие показатели необходимо отслеживать, чтобы определить эти уровни?

На основе ответов SRE-инженеры могут разработать план контроля соответствия продукта ожиданиям бизнеса и конечных пользователей. Мы также можем использовать этот подход, чтобы узнать, как наши пользователи взаимодействуют с системой и как ведет себя сама система. Это поможет понять, предоставляем ли мы качественный продукт. Давайте начнем с ответов на поставленные вопросы в контексте нашей API-песочницы.

Сначала нам нужна информация о трех вещах:

- соглашении об уровне обслуживания (SLA, service-level agreement);
- целях уровня обслуживания (SLO, service-level objectives);
- показателях уровня обслуживания (SLI, service-level indicators).

Связи между этими показателями представлены на рис. 12.1.

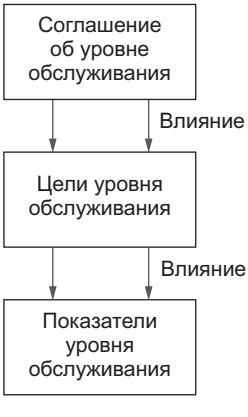

Рис. 12.1. Модель, показывающая взаимосвязь соглашения об уровне обслуживания, целей уровня обслуживания и показателей уровня обслуживания

Ниже мы обсудим каждый из этих показателей более подробно, но главное, помнить, что цель, зафиксированная в рамках более высокого уровня модели, будет влиять на то, что отслеживается на более низком уровне. Попробуем разобраться в этом на примере платформы restful-booker.

12.1.2. Цели уровня обслуживания

Может показаться странным начинать со среднего уровня модели, но поскольку SLO непосредственно управляется и отслеживается командой, это логичное место для начала. К тому же в ситуациях, когда нет четкого SLA, нам приходится начинать с определения SLO.

SLO — это пороговый показатель, который команда устанавливает для определения соответствия продукта ожиданиям по таким характеристикам, как доступность, надежность и оперативность. Пример SLO: наш продукт должен быть доступен как минимум в 95 % случаев в заданный период времени (24 часа, неделя, месяц и т. д.). Такой SLO мы можем использовать для измерения соответствия продукта ожиданиям в процессе его использования. Если установленное значение не достигается, мы можем реагировать соответствующим образом, выявляя и решая проблемы.

SLO сходны с характеристиками качества, которые мы обсуждали в главе 3. Традиционно SLO имеют более выраженную техническую направленность, поскольку определяются путем измерения системных показателей. При этом они, как и характеристики качества, направлены на пользователя и его впечатления от продукта. При определении SLO мы фактически определяем значения характеристик качества.

Поэтому давайте вернемся к работе, проделанной в главе 3, и вспомним характеристики качества, которые мы определили для платформы restful-booker, чтобы посмотреть, можем ли мы определить для них соответствующие SLO. Итак, наша стратегия тестирования направлена на улучшение следующих характеристик качества платформы restful-booker:

- интуитивность;
- завершенность;
- стабильность;
- конфиденциальность;

- доступность;
- настраиваемость.

Давайте выберем одну из этих характеристик и превратим ее в SLO. Что это будет? Некоторые из характеристик в списке сложны для измерения. Например, нет очевидной метрики для интуитивности. Конечно, можно использовать время между запросами в качестве оценки того, насколько легко пользователю ориентироваться в системе, но тогда слишком многие факторы останутся неучтенными. Однако другие характеристики, такие как доступность, измерить проще. Поэтому, ориентируясь на доступность, установим следующую SLO:

> Доступность платформы restful-booker будет составлять 99 % в круглосуточном режиме.

Теперь нужно подумать о том, какие показатели мы будем измерять. Но прежде давайте рассмотрим еще один вариант определения SLO.

12.1.3. Соглашения об уровне обслуживания

SLA — почти то же самое, что и SLO, со следующими оговорками:

- SLA обязательно указывается в договоре с пользователем;
- в случае невыполнения согласованного уровня SLA может налагаться штраф.

Допустим, что мы согласовываем и подписываем SLA с каждым клиентом — владельцем B&B. В этом SLA указывается значение доступности с дополнительными деталями о штрафах. Например:

> Доступность платформы restful-booker будет составлять 95 % в течение месяца. В случае доступности ниже 95 % клиенту будет выплачен штраф в размере $1000.

Очевидно, что это не идеальный пример юридического соглашения, но он подчеркивает несколько важных различий между SLA и SLO. Во-первых, цифра 95 % отличается от 99 %, указанных в SLO. В данном примере SLA 95 % могло быть согласовано с клиентом другим отделом (возможно, юридической или клиентской службой). Установив для команды более высокий целевой порог в 99 %, мы обеспечим запас времени на реакцию в случаях, если значение доступности опустится ниже SLO, прежде чем начнут применяться штрафные санкции.

Баланс между стоимостью и показателями

Более высокий порог для SLO имеет смысл, чтобы располагать временем для реагирования на ситуации, в которых показатели системы падают. Но мы должны учитывать стоимость высокого порога. Поддержание доступности системы на уровне 99 % требует больших затрат времени и ресурсов, а также создает больше потенциальных помех для повседневной работы. При разработке SLO разумно выбрать цель, которая не является запредельно дорогой, но оставляет некоторое пространство для реакции на непредвиденные ситуации.

Как мы уже отметили выше, второе ключевое различие между SLA и SLO — это возможность штрафа в случае невыполнения целей. Штрафы вводятся как способ компенсации клиентам потери дохода и как средство мотивации бизнеса к соблюдению определенных стандартов. Но штрафы также дают нам представление о ценностях клиентов и их понимании качества. Показатели, которые фигурируют в SLA как приоритетные или сопровождаются более высокими штрафами, могут многое рассказать нам о важных характеристиках качества. Мы можем учесть их при тестировании в продакшене и других видах тестирования. Это поможет определить риски с более высоким приоритетом и выбрать мероприятия по тестированию.

В заключение следует отметить, что если SLA определено, то установленные в договоре уровни обслуживания будут определять минимальные значения SLO. Однако инженерные команды должны определять SLO также на основе исследования представлений пользователей о качестве. Когда SLO разработаны, мы можем определить, какие метрики будем отслеживать, чтобы контролировать достижение заданных значений.

12.1.4. Показатели уровня обслуживания

Если SLA и SLO устанавливают, каким значениям показателей должен соответствовать продукт, то SLI используется для отслеживания успеха или неудачи в достижении этих значений. Например, вернемся к нашей SLO:

> Доступность платформы restful-booker будет составлять 99 % в круглосуточном режиме.

Нам нужно разработать способ измерения доступности платформы restful-booker. Для этого доступны следующие возможности:

- перехватывать код состояния каждого запроса, выполненного пользователем;
- записывать результаты пингов для каждого API;
- запрашивать показатели оборудования и оценивать их уровень.

Хотя самым простым подходом может быть регулярное пингование каждого API и запись кодов состояний, нам нужны данные, отражающие опыт пользователя. Именно поэтому мы будем отслеживать коды состояний запросов пользователей.

Мы будем записывать код состояния из каждого ответа на запрос, сохраняя его в течение 24 часов (установленных нашим SLO), чтобы определить, обеспечивается ли доступность 99 %. Таким образом, если доля кодов ошибок или сетевых тайм-аутов в течение 24 часов укажет на снижение показателя ниже 99 %, мы будем знать, что больше не выполняем SLO и рискуем нарушить SLA.

Этот пример дает нам базовое представление о том, что такое SLI, но нужно учитывать целый ряд вещей помимо того, что мы собираемся измерять. Отслеживание кода состояния — это основа для SLI, но также нам может понадобиться следующее:

- *Учет разных местоположений*. Наши экземпляры платформы могут быть развернуты на серверах в разных точках мира. Возможность выявления регионов, где доступ проблематичен, поможет диагностировать проблемы заранее.
- *Анализ прошлых данных*. Хотя контроль значений выбранного показателя позволяет устранять факторы, снижающие доступность, важен и ретроспективный анализ, чтобы понимать тенденции и выявлять системные проблемы. Например, медленное развертывание может вызывать скачки значений доступности.
- *Контроль времени отклика*. API, которому требуется 20 секунд для ответа, не намного лучше, чем API, который не работает. Наш SLA может оговаривать, что время ожидания ответа более 5 секунд приравнивается к недоступности сервиса. Отслеживание времени отклика поможет измерять доступность и диагностировать проблемы (например, узкие места в производительности или негативное влияние сторонних приложений).

Это лишь несколько примеров дополнительных соображений, которые следует принять во внимание при определении SLI. Они важны не только для решения о том, как мы будем контролировать выполнение SLO/SLA, но и для выбора и настройки инструментов.

12.1.5. Что сохранять

Хотя SLI указывают, какие метрики нужно измерять, все равно необходимо тщательно продумать выбор и настройку измерительных средств. Может возникнуть соблазн использовать инструмент, позволяющий собирать широкий спектр метрик. Но нужно принять во внимание, что по мере роста наших систем растет и сбор метрик. Поэтому необходимо убедиться, что мы не столкнемся с такими проблемами, как высокие затраты на управление данными и потеря возможности использовать прошлые данные. Рассмотрим вопросы, связанные с выбором измерительных инструментов.

Структура метрики

Многие библиотеки и фреймворки предлагают ведение журналов, но их структура и объем не всегда оптимальны для конкретных задач. Традиционно журналы используются как средство информирования о действиях или проблемах в системе. Но наша цель — отслеживать информацию о взаимодействии между пользователями и нашей системой. Это еще более сложно при работе с системами, использующими распределенный набор веб-API, каждый из которых отвечает за определенные сервисы. Представьте трудоемкость диагностирования проблемы в системе с более чем 100 веб-API путем просмотра лог-файлов.

Поэтому целесообразно структурировать наши метрики с помощью *событий*. После выполнения каждого запроса фиксируется набор информации. Мы можем регистрировать общие атрибуты, такие как заголовки HTTP-запросов, полезная нагрузка и коды состояния HTTP-ответов. Но также можно сохранять и другую полезную информацию, например уникальные идентификаторы и данные о пользователе. Кроме того, события можно легко связать воедино, что поможет нам проследить путь пользовательского запроса через платформу API, облегчая последующий анализ.

Используя инструмент, который ориентирован на события, а не на составление журналов, мы сможем отслеживать поведение пользователей на нашей платформе.

Где и как долго хранить

Еще один фактор, который необходимо учитывать, — где и как долго мы будем хранить метрики. При отслеживании показателей уровней обслуживания у нас есть выбор: можно хранить данные в течение срока их актуальности для определения уровня обслуживания (например, 24 часа) либо дольше, чтобы иметь

возможность анализировать проблемы или выявлять тенденции за длительный период времени. Также возможен компромисс: объединение данных в наборы, которые позволят проводить анализ в историческом разрезе, но с меньшим объемом информации.

Каждый из этих подходов имеет свои преимущества и недостатки. Объем данных, которые будут генерировать наши платформы API, может быть огромным, и нам придется платить за их хранение и анализ. Кроме того, поскольку мы отслеживаем данные реальных пользователей, необходимо учитывать вопросы конфиденциальности и безопасности. Одним из вариантов является анонимизация и даже снижение объема выборки данных после того, как окно SLO будет завершено, то есть удаление несущественных данных.

Дополнительная информация

У нас может быть четкое представление об основных метриках. Так, в примере с платформой restful-booker мы хотим собирать коды состояния и время отклика. Но достаточно ли только их? Если мы хотим иметь возможность анализа, важно собирать полезную информацию, которую мы не можем получить другими способами. Такие данные, как местоположение пользователя, пользовательские агенты и состояние системы, полезны для анализа проблем. Однако здесь нужен компромисс, схожий с определением объема хранилища: чем больше данных мы отслеживаем, тем больше будут затраты на их хранение и обработку.

Затраты времени и бюджет

Список метрик, которые мы хотим собирать, поможет определить инструментарий, но надо учитывать и другие факторы. Например, если важны возможности длительного хранения и исторического анализа, то следует рассматривать инструменты, которые ориентированы на обеспечение наблюдаемости систем. Другая группа инструментов фокусируется на мониторинге.

Выбор инструментов для сбора метрик и настройки оповещений огромен. Наберите в поисковике *monitoring or observability tools* (инструменты мониторинга или наблюдения), и вы обнаружите множество решений с открытым исходным кодом, а также несколько платных вариантов в широком ценовом диапазоне. Здесь также придется идти на компромиссы. Некоторые инструменты предлагают облачную настройку с легкой интеграцией, но при этом имеют высокую цену. Другие обеспечивают возможность создания собственных экземпляров приложений, но требуют значительных затрат времени на настройку и обслуживание, управление данными и резервное копирование. Поэтому необходимо

определить, сколько мы готовы потратить финансовых и временны́х ресурсов на инструменты для тестирования в продакшене. Учтите, что по мере того как инструменты сбора метрик становятся более сложными, увеличивается время на их освоение. Возможно, потребуются инвестиции в повышение квалификации команды, чтобы коллеги могли эффективно пользоваться этими инструментами, а также забота о том, чтобы эти знания оставались у команды по мере ее роста и смены состава.

Для успешного тестирования в продакшене необходим четкий план. Определение показателей уровней обслуживания и того, как мы собираемся их выполнять, даст четкое направление, которого нужно придерживаться. После этого необходимо определить, что нам нужно для проведения тестирования в продакшене, и изучить возможные варианты. Когда мы определимся с выбором, внедрение тестирования в продакшене должно пойти намного легче.

12.2. НАСТРОЙКА ИНСТРУМЕНТОВ ДЛЯ ПРОВЕДЕНИЯ ТЕСТИРОВАНИЯ В ПРОДАКШЕНЕ

По мере развития концепций и методов, лежащих в основе мониторинга, наблюдений и тестирования в продакшене, расширяются возможности инструментов. Чтобы настроить тестирование в продакшене для платформы restful-booker, мы воспользуемся платформой мониторинга Honeycomb. Honeycomb — популярный инструмент, который позволяет легко подключить средства отслеживания событий в нашем приложении и их анализа. Ядро Honeycomb — SaaS-приложение, к которому мы можем получить доступ через интернет. События фиксируются библиотеками Honeycomb, которые встроены в наше программное обеспечение. Honeycomb обеспечит все необходимое для работы с примерами из этой главы, но при выборе инструментария для своих проектов потратьте время на изучение и других вариантов, исходя из ваших SLI.

12.2.1. Настройка учетной записи Honeycomb

Для наших целей нужен инструмент, способный собирать данные о событиях из API и отправлять их для централизованной обработки и анализа, что и предлагает Honeycomb. Как мы увидим во время настройки, Honeycomb обеспечивает простую интеграцию API, богатый набор запросов и легко настраиваемые оповещения, и все это бесплатно. Конечно, в платных версиях доступен широкий спектр дополнительных функций, но и бесплатный вариант предоставляет нам все необходимое для проведения тестирования в продакшене.

Чтобы начать процесс настройки, нужно создать учетную запись в Honeycomb. Эта учетная запись будет центром, куда будут поступать данные о событиях, когда приложение будет запущено. Чтобы создать аккаунт, зайдите на сайт https://honeycomb.io и нажмите Get Started.

После создания учетной записи нужно создать новую команду. Я назвал свою команду rbp и нажал Enter. Когда учетная запись будет настроена, вы увидите свой API-ключ, который будет использоваться для аутентификации данных о событиях из API. Сделайте копию этого ключа. Теперь давайте перейдем к настройке наших API с помощью Honeycomb.

12.2.2. Добавление Honeycomb в API

Как вы, возможно, уже заметили, Honeycomb встроен в `branding` API, `booking` API и `auth` API в кодовой базе платформы restful-booker. Но если мы хотим отслеживать события из других API, нужно интегрировать Honeycomb и в них. В этом разделе мы настроим `room` API.

Одним из преимуществ Honeycomb является простота интеграции с рядом API-фреймворков для Java, JavaScript, C# и др. с открытым исходным кодом.

Во время изучения платформы restful-booker мы узнали, что API построены с использованием SpringBoot. Это отличная новость, потому что Honeycomb имеет простую в использовании библиотеку для соединения Honeycomb и SpringBoot API. Все, что нам нужно сделать, это добавить следующую зависимость в файл pom.xml нашего `room` API:

```
<dependency>
    <groupId>io.honeycomb.beeline</groupId>
    <artifactId>beeline-spring-boot-starter</artifactId>
    <version>1.5.1</version>
</dependency>
```

ПРИМЕЧАНИЕ Обновите версию до последней, если это необходимо.

Библиотека `beeline` используется для сбора данных о событиях и отправки их в нашу учетную запись Honeycomb. Поэтому для завершения установки нам нужно настроить ее, добавив в файл application.properties следующее:

```
honeycomb.beeline.enabled=true
honeycomb.beeline.service-name=rbp-room
honeycomb.beeline.dataset=rbp-room-dataset
honeycomb.beeline.write-key=<APIKEY>
```

Опции `enabled` и `write-key` позволяют включить Honeycomb и сохранить API-ключ, который будет использоваться `beeline` для подключения и отправки событий. Опция `service-name` позволяет добавить имя сервиса к отправляемому событию, чтобы помочь нам определить происхождение события в Honeycomb. Последняя опция, `dataset`, позволяет определить, в какое хранилище данных будут отправляться наши события. Мы можем выбрать одно хранилище данных для всех событий из всех API или же свое хранилище данных для каждого API. Выбор подхода определяет то, как мы хотим искать данные и какие SLI были установлены.

Не сохраняйте ключ в коде

Само собой разумеется, что вы не должны сохранять ключи API в своих репозиториях. Лучше отправить ключ записи Honeycomb в ваш API при развертывании приложения с помощью переменных окружения: `Dhoneycomb.beeline.write-key=$HONEYCOMB_API_KEY`.

Настроив подключение к Honeycomb, вы можете запустить `room` API в IDE или собрать его с помощью команды `mvn clean install`, а затем запустить с помощью `java -jar java -jar restful-booker-platform-room-{versionnumber}.SNAPSHOT.jar`. API загрузится и автоматически установит соединение с Honeycomb, готовый к отправке событий.

Чтобы проверить интеграцию, откройте новое окно терминала и выполните следующий запрос несколько раз, чтобы сгенерировать несколько событий:

```
curl http://localhost:3001/room/
```

После генерации нескольких событий вернитесь в панель управления на сайте https://ui.honeycomb.io/ и либо войдите в систему, либо обновите страницу. Вы увидите, что главная страница обновилась и показывает не ключ API, а метрики событий. Это означает, что мы готовы к добавлению триггеров на основе SLI.

GraphQL или REST? Все они — события!

Еще одним преимуществом Honeycomb является то, что независимо от структуры вашего HTTP-запроса, будь то REST или GraphQL, Honeycomb будет фиксировать их как события. Такие инструменты, как Honeycomb, позволяют одинаково реализовать тестирование в продакшене API разной архитектуры.

Упражнение

Когда `room` API настроен на работу с Honeycomb, добавьте интеграцию с Honeycomb в `report` API и `message` API. Вы также можете обновить настройки других API, чтобы все они обращались к вашей системе Honeycomb.

12.2.3. Расширенные запросы

Теперь мы можем запрашивать данные, чтобы узнать больше о событиях, и создавать предупреждения, называемые в Honeycomb триггерами. Последние информируют нас о выходе показателей за границы, определенные SLO. Но сначала ознакомимся с настройкой событий запроса, которые можно будет превратить в триггеры.

Как мы узнали ранее, все события, отправленные из наших API, хранятся в массивах данных. Чтобы запросить события, сначала нужно выбрать массив данных. Если мы решили добавить все события в один массив данных, то для начала выберите New Query. Если же у нас несколько массивов данных, мы можем просмотреть их, выбрав пункт меню Datasets, а затем выбрав массив данных, который мы хотим запросить. При любом способе перехода на страницу запроса появится форма, показанная на рис. 12.2.

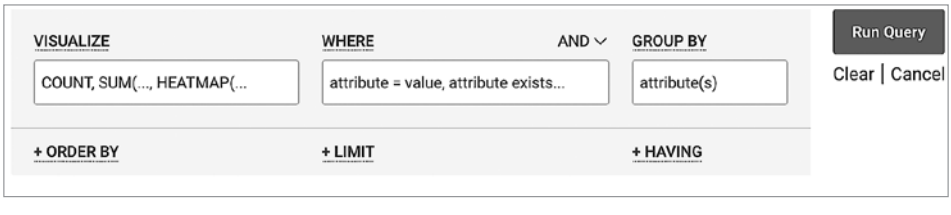

Рис. 12.2. Инструмент Honeycomb, который мы используем для запроса событий

Мы можем использовать этот инструмент для построения запросов по событиям. Например, если нас интересуют данные по кодам состояния за определенный промежуток времени, мы можем выполнить следующий запрос:

- Visualize: `COUNT`
- Group by: `response.status_code`

Нажав кнопку Run Query, вы увидите количество ответов по каждому коду статуса, который был отправлен в течение последних двух часов (рис. 12.3).

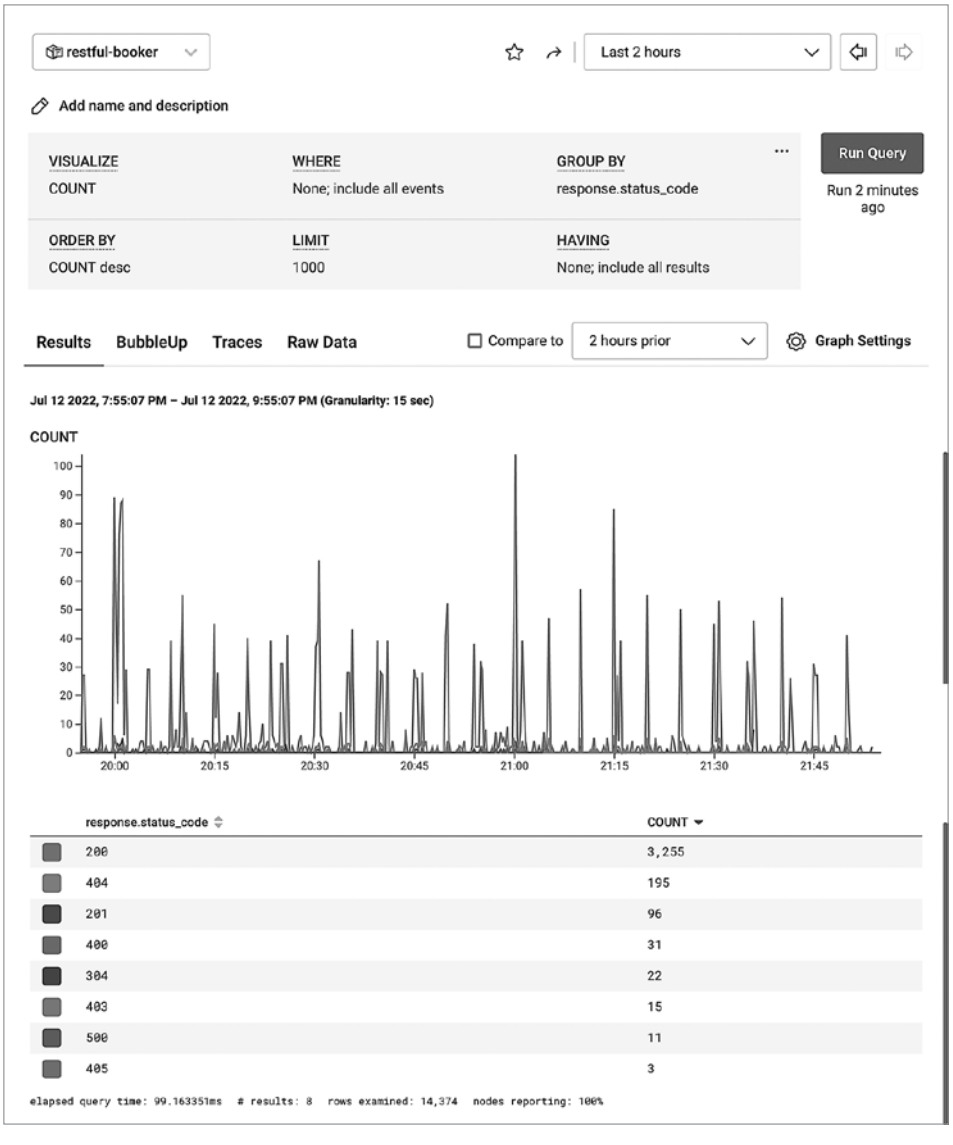

Рис. 12.3. Результаты запроса кодов состояния HTTP-ответов за последние два часа

Если мы сделаем шаг назад к пустой форме запроса и нажмем на поле Visualize, то увидим различные методы анализа наших событий, включая средние значения, процентили и тепловые карты.

Разнообразие возможностей запросов может быть как проклятием, так и благословением. Если существует несколько вариантов создания запросов для наших SLO, то необходимо тщательно продумать выбор. Давайте вспомним SLO для платформы restful-booker:

> Доступность платформы restful-booker будет составлять 99 % в круглосуточном режиме.

Использование текущего запроса подсчета кодов состояния не сработает, поскольку нас интересует соотношение количеств кодов ошибок и кодов успеха, которое и определяет оценку доступности. Мы можем создать триггер, который будет срабатывать каждый раз, когда количество кодов ошибок превышает определенное число, например 100. Но на эти 100 кодов ошибок может приходиться 10 000 кодов успеха, а значит, фактическая доступность все еще выше 99 %, и мы потеряем время, пытаясь решить проблему, которой нет. Если мы составим запросы неправильно, то можем пострадать от ложноотрицательных результатов не меньше, чем от ложноположительных.

Давайте рассмотрим, как создать более надежный запрос для нашего триггера. Honeycomb не предлагает готового способа запроса процента кодов ошибок за определенное время (по крайней мере, бесплатно), поэтому нам потребуется создать собственную функцию запроса, называемую производным столбцом (Derived Column).

Прежде чем перейти к рассмотрению производных столбцов, давайте определим их местонахождение в Honeycomb. Щелкните слева на пункте меню Datasets, а затем на кнопке Settings для созданного нами массива данных. В настройках массива данных перейдите на вкладку Schema, чтобы загрузить раздел Derived Columns. Наконец, нажмите кнопку Add new Derived Column, чтобы открыть редактор.

Производные столбцы позволяют создавать выражения для обработки данных о событиях. Результаты обработки затем можно запрашивать. Например, если нам нужно узнать, какой процент от всех кодов состояния составляют коды ошибок в определенный период времени, то в поле Function мы добавим выражение:

```
IF( GTE( $response.statuscode, 500), 1, 0)
```

В поле Column alias назовем эту функцию `http_error_rate`. Мы создали производный столбец, который при использовании в запросе будет делать следующее:

1. Проверять код статуса ответа каждого события.
2. Если код состояния больше или равен 500, возвращать 1, если нет — 0.

Эта функция дает нам только двоичную оценку, но мы можем использовать ее в запросе. Сохраним производный столбец и создадим новый запрос со следующими настройками:

VISUALIZE: `AVG(http_error_rate)`

Обратите внимание, что имя параметра `http_error_rate` в запросе совпадает с названием производного столбца. Мы вычисляем оценку для всех единиц и нулей, которые были получены от кодов состояний событий. Если она равна 0, то ошибок нет; если 1, то все ответы являются ошибками. Значение оценки лежит в диапазоне между 1 и 0, что дает нам метрику, на основе которой мы создаем триггер.

12.2.4. Создание триггеров SLO

Теперь, когда у нас есть запрос и производный столбец, создать триггер относительно просто. Для начала щелкните на пункте меню Triggers и выберите New Trigger. Вам будет предложено выбрать массив данных, для которого создается триггер. После этого для создания триггера необходимо указать следующее:

- *Имя (Name)*. Если мы планируем использовать несколько запросов, подойдет простое лаконичное название, например `Error Rate >= 1%.`
- *Описание (Description)*. Honeycomb позволяет работать с несколькими учетными записями, поэтому стоит завести привычку описывать оповещения подробно, чтобы могли разобраться коллеги.
- *Состояние (State)*. Необходимо установить значение `Enabled`. Эту опцию можно использовать и для отключения триггера, если он временно не нужен.
- *Запрос (Query)*. Здесь мы указываем наш запрос `AVG(http_error_rate)`.
- *Порог (Threshold)* определяет условие отправки оповещения о том, что мы больше не выполняем SLO. Согласно нашему SLO, мы хотим, чтобы доступность составляла 99 %, поэтому должны установить порог >= 0,01.
- *Получатели (Recipients)* — это те, кого мы хотим уведомить. Уведомление может быть отправлено по электронной почте или через интегрированные инструменты, такие как Slack. Пока что добавим адрес электронной почты, чтобы протестировать триггер.

Сохраним наш триггер, который запустит запланированный запрос и анализ данных о событиях. Если количество ошибок превысит 0,01 % от числа ответов,

будет отправлено оповещение по электронной почте (мы можем проверить эту функцию, нажав кнопку Test). Теперь у нас есть цикл обратной связи, позволяющий реагировать на события, которые могут повлиять на нашу способность соответствовать SLO.

12.3. ДАЛЬНЕЙШЕЕ ТЕСТИРОВАНИЕ В ПРОДАКШЕНЕ

Созданный инструментарий помогает получить реальное представление о том, что ощущает пользователь в конкретный момент времени при работе с нашей системой. Полученная информация помогает определить, достигаем ли мы поставленных целей, и оценить качество продукта. Это полезно для нашей стратегии тестирования и дает много других преимуществ. С помощью таких инструментов, как Honeycomb, мы получаем огромное количество информации о системе, которую можно объединить с результатами других видов тестирования, чтобы узнать больше о нашем продукте и его пользователях. Давайте вкратце рассмотрим некоторые из этих возможностей.

12.3.1. Тестирование с применением синтетических пользователей

Одним из преимуществ использования такого инструмента, как Honeycomb, является то, что события, которые он фиксирует, могут исходить как от реального, так и от синтетического пользователя. Синтетический пользователь — это термин, используемый для обозначения действий в производственной среде, которые имитируют взаимодействие реального пользователя с продуктом. Они могут быть выполнены либо членом команды, либо, как это чаще всего делается, с помощью автоматизированных инструментов.

Например, при настройке `room` API мы можем использовать автоматизацию для выполнения серии вызовов API, которые воспроизводят пользовательский поток создания комнаты и последующего обновления ее деталей. Если мы запустим эту автоматизацию в производственной среде, в то время как Honeycomb принимает события, то сможем наблюдать, срабатывают ли предупреждения. Синтетические пользователи могут быть задействованы, чтобы помочь устранить неожиданные проблемы и повысить нашу уверенность в том, что производственные системы работают на должном уровне.

Синтетические пользователи по сравнению с дымовыми тестами (smoke tests)

Другая техника, используемая в производственных средах, — дымовое тестирование, при котором проверяются наиболее распространенные потоки через систему, чтобы убедиться, что продукт функционирует на высоком уровне. Хотя действия при дымовых тестах и тестировании с синтетическими пользователями очень похожи, цели этих подходов различны.

Дымовое тестирование делает акцент на интеграционных рисках. Образно говоря, дымовое тестирование — это пускание дыма в трубу и наблюдение за тем, где дым выходит, чтобы обнаружить дыры. Тот же принцип применяется при дымовом тестировании производственной системы. Цель состоит в том, чтобы испытать каждую часть системы, чтобы убедиться, что все они работают правильно и интегрированы друг с другом. Это отличается от тестирования с синтетическими пользователями, которое направлено на генерацию событий искусственным образом, чтобы убедиться, что мы выполняем SLO.

Как мы уже обсуждали ранее, наш контекст является ключевым фактором, определяющим, можно ли воспользоваться определенной техникой. Чтобы лучше понять, целесообразно ли использование синтетических пользователей, следует ответить на следующие вопросы:

- *Затрагивает ли тестирование реальных пользователей?* Нужно убедиться, что наши тесты не повлияют на использование системы реальными пользователями. Например, нам не нужно появление в общем доступе тестовых страниц со служебным текстом. Еще хуже разочарование реальных пользователей, обнаруживших, что их действия заблокированы нашим тестированием (например, забронирован последний номер на определенную дату). Чтобы избежать этого, необходим контроль, в первую очередь над данными, с которыми мы взаимодействуем. Если мы не можем легко настраивать и удалять данные или безопасно манипулировать доступом к информации, то использование синтетических пользователей нежелательно.
- *Можете ли вы гарантировать, что тестовые данные не влияют на безопасность?* Работа в живой среде может означать необходимость соблюдения жесткого контроля безопасности. Не следует проводить тестирование, которое может привести к компроментации или утечке данных реальных пользователей, например, если мы воспроизводим пользователя-администратора, который имеет доступ к конфиденциальной информации.

Кроме того, возможны проблемы безопасности инфраструктуры, которые не позволят нам получить доступ к производственной среде так же, как к тестовой.

- *Можете ли вы отличить тестовые события от событий реального пользователя?* Следует рассмотреть и возможность отделить тестовые события от событий реальных пользователей. С помощью таких инструментов, как Honeycomb, можно фильтровать широкий спектр атрибутов событий. Например, если мы можем фильтровать пользовательские агенты, то сможем использовать фальшивый пользовательский агент для наших синтетических пользователей, чтобы отделить реальные события от тестовых. Это поможет диагностировать проблемы и проанализировать результаты, поскольку мы не хотим, чтобы наши тестовые события искажали результаты таких мероприятий, как A/B-тестирование, о котором мы узнаем далее.

Перечисленные соображения показывают, что для создания и использования синтетических пользователей требуется хорошее планирование. Это не так просто, как применять существующую автоматизацию в производственной системе или выполнять другие мероприятия тестирования, пока у нас включены такие инструменты, как Honeycomb.

Хотя поведение синтетических пользователей основано на предположениях о том, как ведут себя реальные пользователи, первые не дадут нам точного представления о взаимодействии с продуктом реальных пользователей. Мы должны рассматривать возможность использования этого подхода в сочетании с отслеживанием реальных событий от пользователей, чтобы получать качественную обратную связь.

12.3.2. Тестирование гипотез

Анализируя данные о событиях и сопоставляя их с SLO, мы определяем, соответствуют ли показатели продукта нашим ожиданиям и ожиданиям клиентов. Однако из собранных статистических данных можно узнать гораздо больше. При правильном подходе данные о событиях можно проанализировать, чтобы узнать больше о поведении пользователей и их отношении к продукту. Для этого необходимо сформулировать четкие гипотезы, на которые мы хотим получить ответы. Затем проводится эксперимент для проверки гипотезы с помощью таких методов, как A/B-тестирование, которое поможет сравнить поведение разных пользователей в зависимости от того, какие функции мы им предоставляем.

Чтобы проиллюстрировать этот подход, предположим, что мы хотим выяснить, как пользователи реагируют на новую функцию, в которой конечная точка `GET /room/` начинает постранично отображать списки комнат, вместо того чтобы показывать их все в одном списке. Мы можем включить функцию постраничного просмотра таким образом, чтобы, например, 50 % пользователей видели ее (группа A), а 50 % — нет (группа B). Затем мы определим продолжительность эксперимента, а после его завершения проанализируем события в каждой группе. Нам понадобится сравнить величины трафика в зависимости от варианта отображения, отправку пользователями неожиданных запросов, которые могут привести к ошибкам, и т. д. Цель — узнать больше о том, как пользователи относятся к нововведениям, которые мы сделали для улучшения качества продукта. Если новая функция будет воспринята негативно, значит, ее необходимо доработать или забыть о ней.

Использование методов A/B-тестирования с помощью таких инструментов, как Honeycomb, поможет узнать больше о наших пользователях. Но, как и другие рассмотренные виды тестирования, они требуют предварительного планирования. Вот некоторые замечания относительно использования A/B-тестирования и измерения результатов:

- *Планирование*. Хотя мы не можем точно предсказать результаты A/B-тестирования, важно иметь четкое представление о том, что считать успехом, а что нет. Прежде чем приступать к экспериментам, необходимо определить, что мы хотим узнать в первую очередь. В противном случае результаты не принесут пользы.
- *Настройка*. Такие методы, как A/B-тестирование, требуют, чтобы продукт можно было настроить на различные состояния. Существует множество инструментов для A/B-тестирования, которые помогут в настройке, но она все равно потребует опыта и времени.

Проще говоря, для успешного A/B-тестирования нам необходимо четко определить гипотезу и план ее проверки и только затем провести тестирование. Если каждый шаг будет тщательно продуман, то полученная информация даст важные сведения, которые другие виды тестирования вряд ли смогут предоставить.

В начале этой главы мы упомянули книгу «Бизнес с нуля» Эрика Риса, в которой объясняется, что для успеха бизнеса необходимо понимать, что значит качество для клиента, а неспособность это понять приводит к пустым затратам и возможному провалу. Эта идея созвучна целям тестирования, которое должно быть направлено на помощь командам в создании высококачественных

продуктов. Именно поэтому тестирование в продакшене, если оно выполнено хорошо, может вознаградить нас лучшим пониманием того, чего хотят клиенты или пользователи. Ни одна другая техника, которую мы рассматривали в этой книге, таких данных не принесет. Разумеется, это не означает, что все остальное, чему мы научились, становится ненужным. Но как часть целостной стратегии, тестирование в продакшене предлагает возможности, которые фокусируются на реальных взаимодействиях между нашими системами и пользователями.

12.4. РАСШИРЕНИЕ СТРАТЕГИИ ЗА СЧЕТ ТЕСТИРОВАНИЯ В ПРОДАКШЕНЕ

В этой книге мы рассмотрели широкий спектр видов тестирования, и объединяет их одно: направленность на получение информации до того, как новые функции или продукты будут запущены в продакшен. Эти техники помогают разобраться, что мы создаем, и дают уверенность в качестве продукта. Но что, если мы ошиблись и неправильно оценили потребности пользователя? Что, если мы не учли критический риск? Возможно, нас ждет разочарование, когда мы наконец представим продукт пользователям.

Ключ к решению этой проблемы не в том, чтобы удвоить количество тестирований. Нужно признать, что иногда мы проводим бесполезные тестирования, которые в лучшем случае приносят мало пользы, а в худшем дают нам ложное чувство уверенности. Мы уже говорили о том, что число потенциальных объектов для тестирования всегда будет превышать то, что можно реально протестировать. Поэтому возможны бесполезные тестирования и пробелы в наших представлениях. Более того, в той или иной степени такие ситуации неизбежны, так что с этим фактом надо смириться и разработать стратегию, которая поможет вернуться на правильный курс.

Тестирование в продакшене эффективно, поскольку позволяет одновременно изучать области представлений (воображение) и реализации, как показано на рис. 12.4.

Используя тестирование в продакшене, мы можем сделать следующее:

- *Улучшить наше понимание области представлений, наблюдая за тем, как наши пользователи взаимодействуют с нашим продуктом.* Например, если новая функция сложна в использовании или неудобна, мы можем установить это по метрикам, характеризующим ее использование.

- *Улучшить наше понимание области реализации, фиксируя проблемы, которые возникают в процессе использования нашего продукта.* Например, если после релиза отмечается замедление работы, нам может понадобиться время, чтобы выяснить его причины.

Рис. 12.4. Модель стратегии тестирования показывает, что тестирование в продакшене позволяет исследовать работу приложения в реальных сценариях (реализация) и взаимодействие пользователей с ним (воображение)

Тестирование в продакшене выглядит рискованным предприятием, и это может отпугнуть команды от внедрения такого подхода к тестированию. Но узнавая больше об использовании продукта, мы можем лучше понимать отношение пользователей к нашей работе и то, как продукт ведет себя в реальных условиях, а это поможет улучшить качество.

ИТОГИ

- Тестирование в продакшене — это наблюдение за тем, как пользователи взаимодействуют с нашей системой и как ведет себя система. Цель — лучше понять пользователей.
- Тестирование в продакшене помогает выявлять проблемы, упущенные при других видах тестирования, которые мы можем решить и проанализировать для улучшения этих других видов.
- Мы можем использовать методы, созданные SRE-инженерами, измеряя поведение системы и сравнивая ее с ожиданиями, чтобы оценить качество нашего продукта.

- Мы устанавливаем измеряемые параметры на основе характеристик уровня обслуживания.
- Цели уровня обслуживания используются для определения конкретных моделей поведения, которым должна соответствовать система.
- Соглашения об уровне обслуживания схожи с целями уровня обслуживания в том, что они устанавливают уровни ожиданий, но также связаны с финансовыми штрафами и обычно определяются бизнесом.
- Показатели уровня обслуживания используются для контроля качества работы продукта. Они также помогают определить, какие инструменты нам необходимы для оценки наших систем.
- Мы можем интегрировать такие инструменты, как Honeycomb, в наши API, чтобы отслеживать данные о событиях.
- Мы создаем запросы в Honeycomb для анализа данных, а также для создания триггеров, которые информируют нас, когда мы перестаем соответствовать SLO.
- Тестирование в продакшене позволяет опробовать такие подходы, как использование синтетических пользователей и A/B-тестирование.

Приложение. Установка платформы API-песочницы

НАСТРОЙКА ПЛАТФОРМЫ RESTFUL-BOOKER

Для лучшего освоения материала мы будем рассматривать практические примеры тестирования API-песочницы — платформы restful-booker. Это платформа веб-API, которая была специально разработана в качестве учебного пособия с помощью Java и JavaScript. Исходный код можно найти на сайте http://mng.bz/AVWp. Чтобы установить платформу restful-booker, вам понадобятся следующие инструменты:

- Java SDK;
- Maven;
- NodeJS LTS.

Требования к версиям указаны в документе README. Они регулярно обновляются.

Зачем нужен NodeJS?

JavaScript-компонент платформы использовался для создания пользовательского интерфейса платформы restful-booker. Эта часть приложения практически не будет обсуждаться в книге (поскольку мы сосредоточимся на веб-API продукта), но она необходима для его запуска.

Чтобы собрать приложение локально, нужно запустить команду `bash build_locally.sh` в Linux или Mac и `build_locally.cmd` в Windows. При первом запуске сборка может занять некоторое время из-за загрузки зависимостей. Когда приложение будет собрано, вы сможете получить к нему доступ по адресу http://localhost:8080. В дальнейшем запускайте платформу командой `bash run_locally.sh` в Linux или Mac и `run_locally.cmd` в Windows.

Кроме того, общедоступная версия платформы restful-booker представлена по адресу https://automationintesting.online.

Марк Винтерингем

Тестирование веб-API

Перевел с английского А. Гаврилов

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*Д. Гудилин*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

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: 02.2024. 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 23.01.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 1000. Заказ 0000.