

O'REILLY®

Фулстек тестирование

создаем качественные программы



SPRINT
book

Гаятри Мохан
Предисловие Ребекки Парсонс

Full Stack Testing

*A Practical Guide for Delivering
High Quality Software*

Gayathri Mohan

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Фулстек-тестирование

*создаем качественные
программы*

Гаятри Мохан

SPRiNT
book 2024

Выпущено
при поддержке

КРОК

ББК 32.973.26-018.2
УДК 004.4
М86

Гаятри Мохан

Г24 Фулстек-тестирование. — Астана: «Спринт Бук», 2024. — 416 с.: ил.
ISBN 978-601-08-4035-5

Тестирование — важнейшая задача для создания высококачественного программного обеспечения. Разработчики и инженеры по качеству найдут в книге всеобъемлющий материал по тестированию в десяти различных категориях. Познакомьтесь с соответствующими стратегиями и понятиями и получите практические знания, применимые как при разработке, так и при тестировании мобильных и веб-приложений.

Автор предлагает примеры использования более чем 40 инструментов, которые вы сможете немедленно опробовать. Профессионалы и новички получат навыки тестирования производительности, защищенности и доступности, а также поближе познакомятся с особенностями исследовательского, межфункционального тестирования, тестирования данных и мобильных приложений, автоматизации тестирования, и многими другими вопросами, связанными с тестированием. Руководство поможет вам справиться со сложной задачей поддержания качества программного обеспечения в процессе его разработки.

ББК 32.973.2-018-07
УДК 004.415.53

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1098108137 англ.

Authorized Russian translation of the English edition of Full Stack Testing
ISBN 978-1098108137 © 2022 Gayathri Mohan.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-601-08-4035-5

© Перевод на русский язык. ТОО «Спринт Бук», 2024

© Издание на русском языке, оформление. ТОО «Спринт Бук», 2024

Краткое содержание

https://t.me/it_boooks/2

Предисловие	15
Введение	17
Благодарности	21
От издательства	23
Глава 1. Введение в фулстек-тестирование.....	24
Глава 2. Ручное исследовательское тестирование.....	37
Глава 3. Автоматизированное функциональное тестирование	74
Глава 4. Непрерывное тестирование.....	124
Глава 5. Тестирование данных.....	151
Глава 6. Визуальное тестирование.....	187
Глава 7. Тестирование безопасности	213
Глава 8. Тестирование производительности.....	250
Глава 9. Тестирование доступности.....	295
Глава 10. Тестирование межфункциональных требований	319
Глава 11. Тестирование мобильных приложений.....	345
Глава 12. За рамками тестирования	382
Глава 13. Введение в тестирование новых технологий.....	393
Об авторе	411
Иллюстрация на обложке	412

Оглавление

Предисловие	15
Введение	17
Почему я написала эту книгу	17
Кому адресована эта книга	18
Структура издания.....	19
Условные обозначения.....	20
Благодарности.....	21
От издательства	23
О научных редакторах русскоязычного издания.....	23
Глава 1. Введение в фулстек-тестирование.....	24
Фулстек-тестирование для достижения высокого качества	26
Тестирование на ранних этапах разработки ПО	28
Десять навыков фулстек-тестирования	32
Ключевые выводы.....	36
Глава 2. Ручное исследовательское тестирование.....	37
Введение.....	39
Подходы к исследовательскому тестированию	39
Изучение функциональности	48
Стратегия ручного исследовательского тестирования.....	52
Изучение приложения	53
Исследование по частям	55
Поэтапное повторение исследовательского тестирования.....	56
Упражнения.....	57
Тестирование API	58
Тестирование веб-интерфейса.....	65
Перспективы: гигиена тестовой среды	71
Ключевые выводы.....	73

Глава 3. Автоматизированное функциональное тестирование	74
Введение.....	76
Введение в типы микро- и макротестов	76
Стратегия автоматизированного функционального тестирования	82
Упражнения.....	84
Функциональные тесты пользовательского интерфейса.....	85
Сервисные тесты.....	103
Модульные тесты.....	107
Дополнительные инструменты тестирования.....	111
Pact	111
Karate.....	116
Инструменты ИИ и машинного обучения в автоматизированном функциональном тестировании	116
Перспективы.....	119
Антипаттерны, которых нужно сторониться	119
Стопроцентный охват кода автоматизированными тестами	121
Ключевые выводы.....	122
 Глава 4. Непрерывное тестирование.....	124
Введение.....	125
Введение в непрерывную интеграцию	125
Процесс CI/CT/CD	126
Принципы и правила	131
Стратегия непрерывного тестирования.....	133
Преимущества.....	139
Упражнение.....	140
Git	140
Jenkins.....	144
Четыре ключевых показателя	148
Ключевые выводы.....	150
 Глава 5. Тестирование данных.....	151
Введение.....	152
Базы данных.....	154
Кэши.....	159

Системы пакетной обработки.....	160
Потоки событий.....	162
Стратегия тестирования данных	164
Упражнения.....	166
SQL.....	166
JDBC	172
Apache Kafka и ZeroCode.....	174
Дополнительные инструменты тестирования.....	183
Тестовые контейнеры	183
Deequ	184
Ключевые выводы.....	186
Глава 6. Визуальное тестирование.....	187
Введение.....	188
Введение в визуальное тестирование.....	188
Критически важные для проекта/бизнеса варианты применения	190
Стратегия тестирования фронтенда.....	192
Модульные тесты.....	194
Интеграционные/компонентные тесты	194
Скриншот-тесты	195
Функциональные сквозные тесты	196
Визуальные тесты	197
Кросс-браузерное тестирование	197
Тестирование производительности фронтенда	199
Тестирование доступности	200
Упражнения.....	200
BackstopJS	200
Cypress	205
Дополнительные инструменты тестирования.....	208
Applitools Eyes, инструмент на базе искусственного интеллекта	208
Storybook.....	210
Перспективы: проблемы визуального тестирования	211
Ключевые выводы.....	212

Глава 7. Тестирование безопасности	213
Введение.....	215
Распространенные кибератаки	216
Модель угроз STRIDE.....	220
Уязвимости приложений.....	222
Моделирование угроз.....	225
Стратегия тестирования безопасности.....	233
Упражнения.....	237
OWASP Dependency-Check.....	237
OWASP ZAP.....	238
Дополнительные инструменты тестирования.....	245
Плагин Snyk IDE	245
Talisman, обработка события коммита	246
Chrome DevTools и Postman	247
Перспективы: защита должна стать привычкой	248
Ключевые выводы.....	248
Глава 8. Тестирование производительности.....	250
Введение в тестирование производительности бэкенда.....	251
Взаимосвязь производительности, продаж и выходных.....	251
Простые цели производительности	252
Факторы, влияющие на производительность приложений	253
Ключевые показатели производительности	255
Типы тестов производительности	256
Типы режимов нагрузки	258
Этапы тестирования производительности	260
Упражнения.....	264
Шаг 1. Определение целевых показателей эффективности.....	264
Шаг 2. Описание тестовых сценариев.....	265
Шаги 3–5. Подготовка данных, среды и инструментов	266
Шаг 6. Написание тестовых сценариев и их запуск с помощью JMeter	267
Дополнительные инструменты тестирования.....	275
Gatling	275
Apache Benchmark.....	276

Введение в тестирование производительности фронтенда.....	277
Факторы, влияющие на производительность интерфейса.....	279
Модель RAIL	280
Показатели производительности фронтенда	281
Упражнения.....	283
WebPageTest.....	284
Lighthouse	287
Дополнительные инструменты тестирования.....	289
PageSpeed Insights	290
Chrome DevTools	291
Стратегия тестирования производительности	292
Ключевые выводы.....	294
Глава 9. Тестирование доступности.....	295
Введение.....	296
Персонажи пользователей с ограниченными возможностями.....	297
Экосистема доступности.....	298
Пример. Программы чтения с экрана	299
WCAG 2.0: руководящие принципы и уровни	301
Стандарты соответствия уровню А.....	301
Фреймворки с поддержкой специальных возможностей.....	305
Стратегия тестирования доступности	305
Чек-лист доступности в пользовательских историях.....	306
Инструменты автоматизированного тестирования доступности	307
Ручное тестирование.....	308
Упражнения.....	309
WAVE	310
Lighthouse	313
Модуль Lighthouse для Node	314
Дополнительные инструменты тестирования.....	316
Модуль Pa11y CI для Node	316
axe-core	316
Перспективы: доступность как культура.....	317
Ключевые выводы.....	317

Глава 10. Тестирование межфункциональных требований	319
Введение.....	320
Стратегия тестирования CFR.....	323
Функциональность	325
Удобство использования	326
Надежность	327
Производительность	328
Поддерживаемость	328
Другие методы тестирования CFR	329
Хаос-инжиниринг	330
Тестирование архитектуры.....	334
Тестирование инфраструктуры	336
Тестирование соответствия.....	339
Перспективы — эволюционное развитие и испытание временем	342
Ключевые выводы.....	343
Глава 11. Тестирование мобильных приложений	345
Введение.....	346
Введение в мобильный ландшафт.....	346
Архитектура мобильного приложения	352
Стратегия тестирования мобильных приложений.....	353
Ручное исследовательское тестирование	356
Автоматизированное функциональное тестирование.....	356
Тестирование данных	357
Визуальное тестирование	358
Тестирование безопасности.....	358
Тестирование производительности	359
Тестирование доступности	360
Тестирование межфункциональных требований.....	361
Упражнения.....	363
Appium.....	363
Плагин Appium для визуального тестирования.....	370
Дополнительные инструменты тестирования.....	373
Database Inspector в Android Studio	374

Инструменты тестирования производительности	375
Инструменты тестирования безопасности.....	377
Accessibility Scanner	379
Перспективы — пирамида тестирования мобильных приложений.....	380
Ключевые выводы.....	381
Глава 12. За рамками тестирования	382
Основные принципы тестирования	382
Предотвращение дефектов вместо обнаружения	382
Эмпатическое тестирование	384
Тестирование на микро- и макроуровне.....	385
Быстрая обратная связь	385
Постоянная обратная связь	386
Измерение показателей качества	387
Общение и сотрудничество: ключ к высокому качеству.....	389
Навыки межличностного общения помогают формировать мышление, ориентированное на качество.....	390
Заключение	392
Глава 13. Введение в тестирование новых технологий.....	393
Искусственный интеллект и машинное обучение	394
Введение в машинное обучение	394
Тестирование приложений МО.....	396
Блокчейн	398
Введение в концепции блокчейна	399
Тестирование блокчейн-приложений	402
Интернет вещей	403
Введение в пятиуровневую архитектуру Интернета вещей.....	404
Тестирование приложений Интернета вещей	406
Дополненная и виртуальная реальность	408
Тестирование приложений ДР/ВР	409
Об авторе	411
Иллюстрация на обложке	412

Отзывы о книге

Эта книга охватывает множество вопросов — от ручного исследовательского тестирования до разработки стратегий тестирования разного уровня с применением новых технологий — и будет полезна не только начинающим, но и опытным специалистам по анализу качества. Гаятри проделала феноменальную работу, изложив ровно такой объем теоретических основ, какой необходим для представления темы, и дополнила ее практическими примерами, которые вы сможете применять в своих проектах с помощью имеющихся у вас инструментов и фреймворков.

*Бхарани Субраманиам, руководитель
технологического отдела компании
Thoughtworks India*

Обширный и глубокий обзор стратегий и шаблонов тестирования. Теоретические основы различных видов тестирования подкреплены практическими примерами. Книга Гаятри должна лежать на столе у всех, кто занимается разработкой программного обеспечения и, следовательно, должен тестировать его.

*Салим Сиддики, автор книги Learning
Test-Driven Development*

Эта книга познакомит вас с технологиями фулстек-тестирования и поможет усовершенствовать корпоративные процессы, связанные с тестированием программного обеспечения. Я советую прочитать ее всем инженерам, занимающимся вопросами контроля качества, техническим руководителям проектов и архитекторам программного обеспечения. В книге представлены разные пути и подходы к тестированию, которые можно применять в разных прикладных областях в зависимости от бюджета и временных рамок.

*Низяр Акиф Мовсумова,
инженер-программист EPAM Systems*

Термин «*фулстек-разработка*» предполагает, что разработчик должен обладать дополнительными навыками для выполнения своей работы. Точно так же термин «*фулстек-тестирование*» предполагает владение технологиями, процессами и навыками тестирования, необходимыми для получения высококачественного программного обеспечения. Книга «Фулстек-тестирование» Гаятри Мохан подробно освещает эти многогранные темы и рассказывает читателям, как создавать высококачественное программное обеспечение.

*Шринивасан Десикан, адъюнкт-профессор
и автор книги Software Testing: Principles and Practices*

Подобно мудрецам с завязанными глазами, наощупь исследующим слона, чтобы совместными усилиями выяснить, как же он выглядит, книга Гаятри рассматривает тестирование с разных сторон и помогает получить целостную картину. Специализированные методы тестирования, безусловно, дают положительные результаты, но владение всем стеком технологий позволяет улучшить результаты.

*Нил Форд, директор/архитектор/идейный
вдохновитель в Thoughtworks и автор книги
Software Architecture: The Hard Parts¹*

¹ Ричардс М., Форд Н., Дехгани Ж. Современный подход к программной архитектуре. Сложные компромиссы. — СПб.: Питер, 2023.

Предисловие

В последнее время все чаще можно услышать слова «сдвиг влево» (*shift left*), обозначающие перенос некоторого действия влево (то есть раньше) по временной шкале. Мы знаем, почему важно перенести принятие архитектурных решений и реализацию мер защиты на более ранние этапы разработки программного обеспечения, но не менее важно это и для тестирования. Внедрение тестирования на самых ранних этапах разработки программного обеспечения (ПО) снижает стоимость и сложность исправления ошибок, потому что они обнаруживаются ближе к моменту их возникновения, когда у разработчика еще свежи воспоминания и он может быстро понять, что привело к их появлению. Размышляя о таких вещах, как тестирование производительности, мы сначала анализируем тенденции и лишь потом начинаем беспокоиться о конкретных значениях. Такой подход позволяет выявить узкие места, где наблюдается значительное ухудшение производительности. Затем мы можем выяснить, вызвано ли ухудшение использованием алгоритма, не способным обеспечить высокую производительность, или мы просто допустили ошибку, которая привела к снижению производительности.

Раннее тестирование (или сдвиг тестирования влево) означает, что тестированию подвергается неполное ПО, которое еще будет изменяться, тем не менее дополнительная возможность устранения возникающих проблем намного перевешивает затраты на непрерывное тестирование, особенно когда значительная часть тестов выполняется автоматически. Некоторые виды тестирования, такие как исследовательское тестирование, неизбежно выполняются вручную, и все же в коллекции тестов должны быть и автоматизированные тесты.

Тестирование должно быть комплексным и всеобъемлющим, и название книги Гаятри недвусмысленно говорит об этом. Автор делает исчерпывающий обзор всего стека тестирования, включая тестирование производительности, пользовательского интерфейса, контрактов, сквозное функциональное тестирование (end-to-end functional testing), модульное тестирование и даже тестирование доступности. У многих занимающихся тестированием возникает резонный вопрос: как организовать комплексное тестирование с применением всего стека технологий? И эта книга дает ответ на него. Конечно, существует множество книг о тестировании и даже о гибком (agile) тестировании, агитирующих за тестирование на ранних этапах разработки ПО, но в отличие от них в книге Гаятри подробно рассматриваются все аспекты тестирования современного приложения, описываются проблемы, возникающие в каждом аспекте, и рассматриваются принципы и стратегии, применимые к этим аспектам.

Каждый из ее разделов включает набор практических упражнений, демонстрирующих особенности разных видов тестирования. Я понимаю, что конкретные инструменты, представленные в упражнениях, могут развиваться и меняться со временем. Однако это не уменьшает ценности упражнений, потому что они показывают, как использовать инструменты для создания правильных тестов. Упражнения конкретизируют подход к тестированию, инструменты дают возможность экспериментировать с тестами такого рода. Инструменты неизбежно продолжают развиваться, но стратегии тестирования, с которыми вы познакомитесь, будут оставаться актуальными еще долго.

В книге Гаятри представлен широкий диапазон подходов к тестированию, включая статический анализ, стратегии тестирования данных и даже исследовательское тестирование. Учитывая растущую сложность программных систем, роль исследовательского тестирования становится все более важной. Кроме того, тестированию безопасности отведена отдельная глава, потому что все мы знаем, насколько уязвимыми для хакеров могут быть наши системы. Тестированию доступности также посвящена отдельная глава, описывающая, как можно сделать системы более простыми в использовании даже для людей с ограниченными возможностями.

Каждый аспект тестирования требует учитывать возможные ошибки и создавать стратегии тестирования для их выявления. Правильно сконструированный набор тестов различных типов обеспечивает высокую надежность, позволяющую нам уверенно развивать свои программные системы. Книга Гаятри, основанная на приобретенном ею опыте тестирования различных типов систем, поможет профессионалам, занимающимся разработкой ПО, создавать правильные стратегии тестирования и наборы тестов.

*Доктор Ребекка Парсонс, технический
директор Thoughtworks, соавтор книги
Building Evolutionary Architectures¹*

¹ Форд Н., Парсонс Р., Куа П. Эволюционная архитектура. Поддержка непрерывных изменений. — СПб.: Питер, 2018.

Введение

В индустрии программного обеспечения редкому специалисту вне зависимости от его роли удастся избежать участия в тестировании, потому что оно стало неотъемлемым аспектом разработки ПО, вплетенным в каждый этап этого процесса. В наше время повсеместного внедрения цифровизации, когда различные мобильные и веб-приложения прочно вошли в повседневную жизнь людей, тестирование по различным параметрам качества стало обязательным.

Рассматривая тестирование как отдельную дисциплину, можно заметить, что на протяжении многих десятилетий существования оно развивалось по собственной траектории, расширялось и вбирало новые практики и методологии, фреймворки и инструменты. Ручное тестирование превратилось в ручное исследовательское тестирование и в настоящее время остается фундаментальной частью дисциплины тестирования. Между тем расширение применения автоматизированного тестирования в сочетании с распространением практик непрерывной интеграции и непрерывного развертывания (continuous integration/continuous deployment, CI/CD) привело к стремительному увеличению получаемой от него выгоды. Автоматизированное тестирование межфункциональных требований, таких как производительность, безопасность и надежность, стало совершенно необходимым для получения полноценной обратной связи и непрерывной доставки высококачественного программного обеспечения. Вот почему фулстек-тестирование считается в отрасли желательной специализацией. Я полагаю, что вы, взявшие в руки эту книгу, хотите овладеть приемами фулстек-тестирования, чтобы создавать высококачественное ПО. Спасибо вам за эти устремления и добро пожаловать на борт!

Почему я написала эту книгу

Многие эксперты по тестированию могли написать эту книгу до меня. Но, может быть, им помешали занятость или отсутствие желания. Как бы то ни было, мне выпала такая возможность, и я благодарна судьбе за это! (Хотя, если бы эту книгу написал какой-нибудь другой эксперт, когда я только начала осваивать тестирование, это сэкономило бы мне массу времени: не пришлось бы рыться в сотнях блогов и самой опробовать десятки инструментов, чтобы обрести навыки, накопленные за много лет.)

Занимаясь консультированием, я заметила, что команды, избравшие разумную стратегию тестирования, обычно добивались успеха, тогда как другие в большинстве

случаев терпели неудачу. Например, я видела команды, полагавшиеся исключительно на сквозное тестирование пользовательского интерфейса или только на ручное тестирование и впоследствии вынужденные тратить почти все свое время на обслуживание ПО или устранение постоянно обнаруживаемых дефектов. Некоторые команды проводили только функциональное тестирование, из-за чего в ПО оставались необнаруженными критические нефункциональные проблемы. В целом такие команды характеризовались неудовлетворенностью своей работой, низкой конкурентоспособностью и низким качеством ПО на выходе. Для меня удивительно, что такой перекося в понимании практик тестирования существует и поныне, когда тестирование как дисциплина существует уже несколько десятилетий. Я могу только предположить, что во многом это связано с нехваткой талантливых специалистов по тестированию, а в условиях продолжающейся холодной войны между компаниями — разработчиками ПО, переманивающими друг у друга лучших специалистов, было бы правильно делиться своими знаниями и распространять их как можно шире.

Несмотря на наличие множества руководств по тестированию с применением отдельных инструментов, нет связного описания того, как повысить свою квалификацию с учетом текущих тенденций развития тестирования, в котором приводились бы практические примеры применения множества разных инструментов. А для многих узкоспециализированных направлений, таких как тестирование безопасности и доступности, руководства, ориентированные на начинающих, вообще редкость. Эта книга задумывалась как всеобъемлющий ресурс, который позволит начинающим тестировщикам повысить свою квалификацию до уровня продвинутого новичка во всех направлениях, необходимых для тестирования современных мобильных и веб-приложений.

Под уровнем продвинутого новичка я подразумеваю модель приобретения навыков Дрейфуса, которая определяет пять ступеней, которые преодолевает человек на пути к овладению некоторым навыком: новичок, продвинутый новичок, компетентный, опытный и эксперт. Эта книга написана с амбициозной целью — на практических примерах помочь читателю преодолеть первые две ступени в освоении десяти разных навыков тестирования. Учитывая, что третья ступень — компетентная и ее можно достичь только при наличии обширной практики, эта книга проведет своих читателей настолько далеко, насколько возможно!

Кому адресована эта книга

В первую очередь эта книга адресована новичкам в тестировании ПО, а также специалистам по тестированию, желающим расширить свои знания. Однако она может оказаться полезной всем, кто работает в сфере разработки ПО и чьи обя-

занности пересекаются с тестированием, например разработчикам приложений или DevOps-инженерам. В любом случае главным требованием является наличие некоторых знаний в области программирования, особенно на Java, потому что в книге присутствуют практические упражнения на Java и в некоторых местах на JavaScript. Кроме того, если вы новичок в индустрии ПО, я бы рекомендовала сначала прочитать о процессах разработки ПО в таких методологиях, как водопадная модель или Agile, а потом браться за эту книгу.

Структура издания

Книга начинается с введения в фулстек-тестирование и подробно описывает десять навыков, необходимых для создания высококачественных мобильных и веб-приложений. После знакомства с основами вас ждут десять независимых глав, посвященных отдельным навыкам. Все они организованы одинаково.

- Сначала в разделе «Введение» обсуждаются темы, определяющие контекст. Если вы пока не владеете обсуждаемым навыком, то здесь узнаете, какие умения он включает в себя, а также почему и где применяется.
- Далее следует раздел, посвященный стратегии, где подробно описывается применение навыка в конкретной ситуации.
- В следующем разделе, «Упражнения», читатель найдет пошаговые инструкции по овладению навыком с использованием нескольких инструментов.
- В некоторых главах имеется раздел «Дополнительные инструменты», в котором параллельно обсуждаются инструменты, аналогичные представленным в разделе «Упражнения» или другие, которые могут принести пользу читателю и обогатить понимание этого навыка.
- Наконец, в некоторых главах вы найдете описание моей точки зрения, основанной на личных наблюдениях и опыте, и ключевые выводы, представляющие собой краткий обзор уроков из этой главы.

За десятью главами, посвященными развитию навыков, идет глава, в которой рассказывается, куда двигаться дальше, опираясь на основные принципы и вновь приобретенные навыки тестирования. Для читателей-энтузиастов есть также дополнительная глава, представляющая собой введение в тестирование новых технологий. В ней представлен краткий обзор приемов тестирования четырех новых технологий — искусственного интеллекта (ИИ) и машинного обучения, блокчейна, Интернета вещей и дополненной/виртуальной реальности, — чтобы помочь читателям начать изучать приемы тестирования в этих областях.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины или важные понятия.

Моноширинный шрифт

Применяется для листингов программ, а также используется внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширения.

Моноширинный жирный шрифт

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

Моноширинный курсив

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

Шрифт без засечек

Применяется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса, каталогов.



Этот рисунок указывает на совет или предложение.



Этот рисунок обозначает общее примечание.



Этот рисунок указывает на предупреждение.

Благодарности

В начале своей карьеры даже в самых смелых мечтах я и помыслить не могла о написании полноценной технической книги, тем более для O'Reilly! Но вдохновение, мотивация и благоприятная среда, сложившаяся в Thoughtworks, привели меня на этот путь, и я невероятно благодарна судьбе за то, что оказалась в таком замечательном коллективе увлеченных специалистов и лидеров, вдохновляющих своим примером. Я хотела бы выразить свою признательность и благодарность за поддержку, которую получила от некоторых замечательных сотрудников Thoughtworks: Прасанна Пендсе призывал всех вокруг ставить перед собой высокие цели, а когда я настроилась на это, поддерживал меня до конца; Бхарани Субраманиан тесно сотрудничал со мной до завершения книги и делился своими идеями, помогавшими в работе над всеми главами; Паллави Вадламани, скорее близкий друг, чем коллега, также тесно сотрудничал со мной с самого начала и вычитывал каждую главу. Сатиш Вишванатан, Киф Моррис, Шрирам Нараян, Нил Форд и Судхир Тивари — вот лишь немногие из тех, кто поддерживал меня на разных этапах работы над книгой. Поистине бесценно, когда такие опытные специалисты делятся своими мудрыми и своевременными советами! Я также хотела бы выразить особую благодарность доктору Ребекке Парсонс, техническому директору Thoughtworks и моему образцу для подражания, которая написала предисловие и была настолько любезна, что вызвалась просмотреть главы на стадии рукописи. Какой еще поддержки я могла бы просить от организации?!

Огромное спасибо команде O'Reilly, особенно Джилл Леонард и Мелиссе Даффилд, за организацию процесса выпуска книги, а также техническим рецензентам Крису Нортвуду, Александру Тарлиндеру, Шринивасану Десикану, Салему Сиддики, Яну Молино и Нигяр Мовсумовой, которые дали свои отзывы и помогли довести книгу до того состояния, в котором она находится сегодня.

Я также хочу засвидетельствовать свою исключительную признательность и благодарность моему многолетнему наставнику Дхивье Арунагири, который несколько лет укреплял во мне уверенность в себе и помогал строить карьеру, а также друзьям, бывшим для меня отдушиной всякий раз, когда я выбивалась из сил из-за работы и семейных обязанностей в условиях пандемии. Кроме того, пользуясь предоставившейся возможностью, выражаю свою искреннюю любовь и признательность моим родителям, которые всегда подбадривали и поддерживали меня.

И наконец, не могу не упомянуть моего дорогого мужа Маноджу Махалингаму, ставшего для меня источником вдохновения, другом и наставником, без которого эта книга просто не появилась бы. Я хочу посвятить эту книгу ему и любимой дочери Магати Манодж за то, что давали мне столь необходимое пространство и время для работы над этой книгой на протяжении более чем года.

Когда я пишу эти строки, я думаю: как же мне повезло, что меня окружают такая замечательная семья, друзья и коллеги. Большое спасибо всем! Я бесконечно вам благодарна.

От издательства

Мы выражаем огромную благодарность компании «КРОК» за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу

comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научных редакторах русскоязычного издания

Анна Захарова — старший инженер-тестировщик в компании КРОК. Более 15 лет работала ведущим инженером-тестировщиком на проектах арбитражной судебной системы РФ, судов общей юрисдикции, проектах РОССТАТ, проектах ФОМС, интеграционных проектах.

Дмитрий Колфилд — инженер-тестировщик в компании КРОК. Принимал участие в тестировании и поддержке высоконагруженных информационных систем (декстопные и веб-приложения). Также занимался тестированием миграций в нереляционных базах данных для информационных систем, обрабатывающих большие данные.

ГЛАВА 1

Введение в фулстек-тестирование

В современном мире цифровизация необходима для поддержания и развития любого бизнеса. Многие компании ушли далеко вперед в этом направлении, но есть и такие, которые только начинают строить свои цифровые платформы.

Цифровизация помогает бизнесу расширить охват потребителей от местного их круга до глобального масштаба, что способствует росту производства и увеличению доходов. Почти все малые и крупные предприятия в различных секторах экономики, таких как здравоохранение, розничная торговля, туризм, наука, социальные сети, банковское дело и развлечения, расценивают продвижение своих цифровых стратегий как важнейшую меру, предпринимаемую для охвата новых групп клиентов и получения более высокой прибыли.

На этом пути к цифровизации и модернизации решающим фактором становятся инновации. Предприятия, постоянно внедряющие инновации, остаются сильными и процветают на протяжении многих десятилетий. Классический пример — компания Netflix. В самом начале, в 1990-х годах, это был портал онлайн-проката DVD, а в 2007-м она сосредоточилась на онлайн-трансляции, поглотив собственный бизнес по прокату DVD. Позже она начала производить оригинальный контент под названием *Netflix Originals*. По состоянию на конец 2021 года Netflix была крупнейшим онлайн-сервисом потокового вещания (<https://oreil.ly/AyHBL>) с более чем 200 млн подписчиков по всему миру.

Параллельно с инновационным бизнесом развивается и технологическое пространство, стремясь удовлетворить растущие потребности. Прошли те времена, когда люди были готовы стоять в очереди, чтобы купить билеты в кино, ехать в отдаленный магазин, чтобы приобрести специфический продукт, или носить с собой написанный от руки список покупок, чтобы ничего не забыть. Технологии облегчают решение таких повседневных задач. Мы можем сидеть дома, смотреть любимые развлекательные программы и одним нажатием кнопки виртуально примерить новое платье, запланировать доставку продуктов, сварить кофе с помощью голосовой команды и сделать многое другое.

Учитывая быстрые темпы развития технологий, стратегии цифровизации должны быть универсальными, удовлетворять различные потребности клиентов и позволять оставаться конкурентоспособными в данном секторе. Уже недостаточно просто создать веб-сайт — нужно что-то более масштабное. Рассмотрим такие компании, предоставляющие услуги такси, как Uber и Lyft. Они позволяют получить доступ к своим услугам различными способами: через Интернет, мобильные платформы Android и iOS и даже чат-бот WhatsApp (<https://oreil.ly/1ijA9>). Универсальная стратегия цифровизации помогла этим компаниям распространиться по всему миру и перерасти своих конкурентов.

Инновации и гибкость помогают предприятиям добиться успеха в привлечении критической массы клиентов. Но после этого встает задача обеспечения дальнейшего процветания, получения больших доходов и привлечения еще большего числа клиентов. Мы видели, как отраслевые гиганты, такие как Amazon, используют свою клиентскую базу для организации перекрестных продаж товаров и услуг и дальнейшего расширения бизнеса. Компания Amazon, которая на первых порах существовала как книжный интернет-магазин, теперь занимается перекрестными продажами товаров от продуктов питания до электроники, одежды, ювелирных изделий и многого другого, удовлетворяя спрос на потребительские товары практически во всех сегментах рынка.

Почему я упоминаю эти детали в книге по тестированию программного обеспечения? Потому что современная индустрия программного обеспечения удовлетворяет все эти потребности бизнеса, предоставляя инструменты для выработки новых идей, воплощения их в жизнь и масштабирования с целью охвата новых групп клиентов по всему миру. Несомненно, команды разработчиков ПО находятся на переднем крае, особенно когда срочно требуется обеспечить *высокое качество*! Действительно, качество ПО стало важным критерием на современном конкурентном рынке. Компромиссы в отношении качества ПО эквивалентны признанию проигрыша в конкурентной гонке, что подтверждается многими реальными примерами. Например, в октябре 2014 года индийские гиганты электронной коммерции Snapdeal и Flipkart после многомесячной рекламной кампании провели сезонную распродажу. К сожалению, во время нее, в День большого миллиарда, веб-сайт Flipkart (<https://oreil.ly/C20pD>) несколько раз падал из-за огромного наплыва посетителей, в результате чего компания потеряла массу потенциальных клиентов и недосчиталась значительного объема доходов. Точно так же компания Yahoo! не смогла на равных состязаться со своими конкурентами, несмотря на то что была одной из первых в своем сегменте. Случилось это отчасти потому, что ее специалисты не уделили должного внимания качеству поисковой системы (<https://oreil.ly/CiYDd>), а отчасти из-за ущерба, нанесенного бренду недостаточными мерами безопасности. Это привело к крупнейшей утечке данных в истории (<https://oreil.ly/CP5ma>), в результате которой в 2013 году были раскрыты 3 млрд учетных записей пользователей. Такова цена качества ПО в наши дни!

Подобные примеры можно найти по всему миру. Они подтверждают наблюдение о том, что предприятия, несмотря на любые новые идеи, оказываются на крутом и скользком склоне, когда качество ставится под угрозу, потому что клиенты быстро переходят к более надежным конкурентам. Иногда компании вынуждены выходить на рынок как можно раньше, несмотря на недостаточно высокое качество их программного продукта, но они должны осознавать, что тем самым создают себе технический долг, который необходимо погасить раньше, чем конкуренты воспользуются им в своих интересах. Таким образом, можно утверждать, что качество — это важнейшее условие поддержания бизнеса в долгосрочной перспективе, а высокого качества можно достичь, только сочетая профессиональную разработку и тщательное тестирование и уделяя должное внимание каждому аспекту приложения. Чтобы помочь вам встать на этот путь, расскажу в этой главе, что входит в комплексное тестирование типичного мобильного или веб-приложения.

Фулстек-тестирование для достижения высокого качества

Для начала разберемся, что означают слова «*качество ПО*». Когда-то под качеством ПО подразумевалось отсутствие ошибок, но сейчас любой причастный к созданию ПО согласится, что простого отсутствия ошибок уже недостаточно, чтобы назвать ПО качественным. Если попросить конечных пользователей дать определение понятия «качество», то вы услышите в ответ перечисление таких характеристик, как простота использования, привлекательный внешний вид, надежная защита персональных данных, высокая скорость предоставления информации и круглосуточная доступность услуг. Если попросить сделать то же самое компании, то вам расскажут о рентабельности инвестиций, аналитике в реальном времени, нулевом времени простоя, отсутствии привязки к поставщику, масштабируемой инфраструктуре, безопасности данных, соблюдении законодательства и о многом другом. Все эти аспекты определяют качество современного программного обеспечения. Недостатки в любой из этих областей так или иначе влияют на качество, поэтому их необходимо тщательно проверять!

Список требований к качеству выглядит длинным, но у нас есть инструменты и методологии, способные помочь удовлетворить большинство этих требований. Поэтому для достижения высокого качества необходимо знать эти инструменты и, что более важно, уметь применять их в контексте разработки и тестирования. Цель этой книги — помочь вам обрести навыки тестирования, необходимые для создания высококачественных мобильных и веб-приложений.

Если говорить кратко, то тестирование — это практика проверки того, соответствует ли поведение приложения нашим ожиданиям. Чтобы добиться успеха, тестирование необходимо практиковать на микро- и макроуровнях. Оно должно

быть неразрывно связано со всеми деталями приложения. Тестировать нужно каждый метод во всех классах, каждое значение во входных данных, каждое сообщение в логах, каждый код ошибки и т. д. Не менее важно сосредоточиться на макроаспектах, таких как тестирование возможностей и интеграция возможностей, а также сквозное тестирование рабочих сценариев. Но этим все не ограничивается! Необходимо дополнительно протестировать нефункциональные аспекты приложения — безопасность, производительность, доступность, удобство использования и т. д., чтобы достичь конечной цели — создать высококачественное ПО. Если кратко: мы должны провести *полноценное фулстек-тестирование*! Как показано на рис. 1.1, оно включает в себя тестирование не только различных аспектов приложения на каждом уровне (база данных, сервисы и пользовательский интерфейс), но и приложения в целом.



Рис. 1.1. Обзорная картина фулстек-тестирования

Фулстек-тестирование и разработка должны быть неотделимы друг от друга, как два рельса железнодорожного полотна. Чтобы обеспечить высокое качество продукта, мы должны двигаться в обоих направлениях сразу, иначе неизбежно сойдем с рельсов. Например, представьте, что вы пишете небольшой блок кода для приложения электронной коммерции, вычисляющий общую сумму заказа. Следует убедиться, что код правильно вычисляет сумму, выполняется безопасно и параллельно. Если этого не сделать, в железнодорожном полотне могут возникнуть разрывы, и, продолжив разработку, двигаясь по такой фрагментарной линии, вы получите плохую интеграцию и неоптимальную функциональность. Чтобы внедрить тестирование на таком элементарном уровне, командам необходимо перестать думать о нем как об особых действиях, выполняемых после разработки, как было принято раньше. Фулстек-тестирование должно проводиться параллельно с разработкой и практиковаться на протяжении всего цикла поставки, чтобы обеспечить быструю обратную связь. Практика, при которой тестирование начинается на ранних стадиях цикла поставки, называется *ранним тестированием*, или *сдвигом тестирования влево*, и это ключевой принцип, которому нужно следовать при выполнении фулстек-тестирования, чтобы получить хорошие результаты.

Тестирование на ранних этапах разработки ПО

Последовательность действий в традиционном жизненном цикле разработки ПО включает в себя анализ требований, проектирование, разработку и тестирование, причем последнее — в самом конце. Как видно на рис. 1.2, смещение тестирования на ранние этапы разработки ПО для получения высококачественных результатов предполагает перенос действий по тестированию в начало цикла.

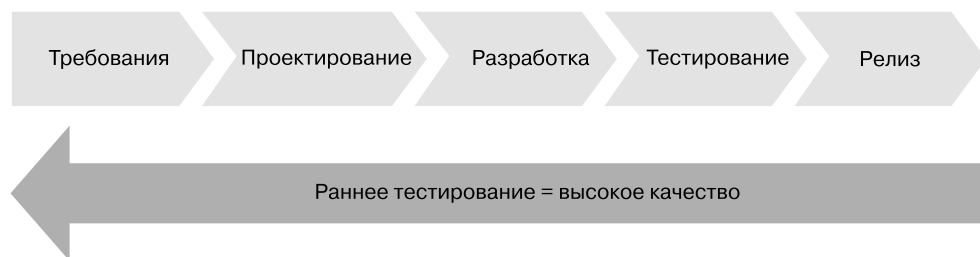


Рис. 1.2. Тестирование на ранних этапах разработки ПО

Рассмотрим аналогию, чтобы лучше понять этот принцип. Представьте, что ваша команда строит дом. Разумно ли сначала завершить строительство и только потом проверять качество? А что, если вы обнаружите, что размеры комнат не соответствуют проекту или внутренние стены недостаточно прочны, чтобы выдержать

нагрузку? Избежать именно таких проблем помогает смещение тестирования на ранние этапы разработки ПО, то есть проверка качества на этапе планирования и на протяжении всего строительства. Такой подход позволяет добиться максимально высокого качества конечного продукта.

Выполнение проверок качества на протяжении всего этапа разработки означает их итеративное повторение после завершения каждого небольшого фрагмента работы, чтобы вносимые изменения не ухудшали качество продукта. В аналогии со строительством дома это означает проверку при возведении каждой стены, чтобы можно было немедленно устранять любые огрехи и недоделки. Для выполнения таких обширных проверок раннее тестирование в значительной степени опирается на автоматизированное тестирование и методы непрерывной интеграции и непрерывного развертывания (CI/CD), когда проверки качества автоматизированы на микро- и макроуровнях и постоянно реализуются для каждого небольшого фрагмента работы на сервере непрерывной интеграции. Только автоматизация может гарантировать постоянное тестирование приложения с минимальными затратами и усилиями по сравнению с ручным тестированием различных аспектов качества.

Чтобы понять, что это означает в контексте ПО, разобьем раннее тестирование на повседневные действия. Рассмотрим команду разработчиков, которая придерживается итеративного цикла разработки, принятого, например, в Agile-разработке. Некоторые проверки качества, которые они могут выполнять на разных этапах, показаны на рис. 1.3.

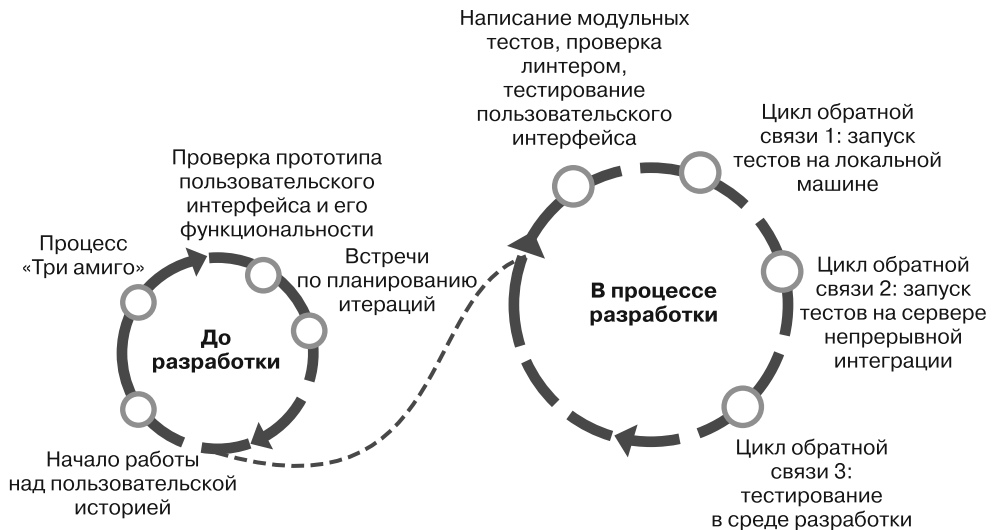


Рис. 1.3. Проверки качества, выполняемые при тестировании на ранних этапах разработки ПО

Диаграмма на рис. 1.3 начинается (*слева*) с определения набора проверок качества, которые команда выполняет до того, как пользовательскую историю можно будет считать готовой к разработке.

- На этапе анализа проходит процесс под названием «Три амиго» (<https://oreil.ly/WFABh>). В ходе него представители бизнеса, разработчики и тестировщики ненадолго собираются вместе и обсуждают функцию, предлагаемую к реализации. Этот процесс направлен на определение точек зрения всех трех ролей, чтобы не упустить из виду интеграцию, крайние случаи и другие бизнес-требования. Это первый шаг к раннему тестированию, когда с самого начала проверяются требования к функции.
- Одновременно представитель бизнеса и дизайнер пользовательского опыта (UX) совместно проектируют и улучшают дизайн приложения.
- По завершении этих двух шагов в начале итерации/спринта проводится *встреча по планированию итерации* (iteration planning meeting, IPM), на которой подробно обсуждаются пользовательские истории, которые предполагается реализовать на этой итерации. Это дает команде возможность еще раз коллективно проверить требования.
- Во время итерации непосредственно перед тем, как пользовательская история будет принята к разработке, начинается *работа над пользовательской историей* — уменьшенная версия процесса «Три амиго», в которой обсуждение фокусируется на требованиях и крайних случаях конкретной пользовательской истории. На этом этапе мы можем с уверенностью сказать, что команда тщательно протестировала/проверила требования.

Точно так же в ходе реализации пользовательской истории внедряются и задействуются следующие проверки качества, обеспечивающие быструю обратную связь.

- Разработчики пишут модульные тесты для каждой истории, добавляют линтеры и плагины для статического анализа кода и интегрируют их в конвейер непрерывной интеграции (CI) для получения постоянной обратной связи.
- В некоторых командах разработчики также пишут функциональные тесты для проверки пользовательского интерфейса и интегрируют их в конвейер CI. В других командах тестировщики пишут такие тесты уже после разработки. Оба подхода считаются обычной практикой.
- Прежде чем отправлять изменения в репозиторий, разработчики запускают набор автоматических тестов на своих локальных компьютерах, создавая первый уровень обратной связи.
- Второй уровень обратной связи подразумевает набор автоматизированных тестов (модульных, сервисов, пользовательского интерфейса и т. д.), которые запускаются на этапе непрерывной интеграции после каждой отправки кода в репозиторий.
- Третий уровень обратной связи образуется в результате процесса, называемого *тестированием в среде разработки*, когда тестировщики и представители

бизнеса проводят быстрый раунд ручного исследовательского тестирования на сервере разработки, чтобы быстро проверить недавно разработанную функциональность.

Благодаря такому пристальному вниманию к обратной связи команда выявит почти половину проблем, которые были бы обнаружены в ходе ручного тестирования после разработки, еще до того, как пользовательская история доберется до самой фазы тестирования. Другими словами, команда просто начинает тестирование на ранних этапах разработки ПО, давая тестировщикам полную свободу исследования пользовательской истории на предмет различных аспектов качества вместо проверки ожидаемого функционального поведения.

Таким образом, раннее тестирование позволяет не только предотвращать (за счет нескольких раундов проверки требований), но и обнаруживать любые дефекты, которые неизбежно появляются на ранней стадии, либо на локальной машине разработчика, либо на сервере CI. Кроме того, такой подход предоставляет тестировщикам возможность глубже изучить различные аспекты качества и тем самым гарантирует поставку высококачественного программного обеспечения.



Экстремальное программирование (extreme programming, XP) — это разновидность Agile-разработки программного обеспечения, которая предполагает раннее тестирование. Желающим глубже изучить методологии и практики XP я могу порекомендовать книгу Кента Бека *Extreme Programming Explained*¹ (Addison-Wesley Professional).

Идея включения тестирования в цикл поставки на ранних стадиях не ограничивается функциональным тестированием приложений. Она вполне применима к тестированию в целом, в том числе к тестированию безопасности, производительности и многим другим видам тестирования. Например, один из многих способов начать тестирование безопасности на ранних этапах — использовать инструмент сканирования перед отправкой кода в репозиторий, такой как Talisman, который проверяет код на наличие уязвимой информации и сообщает о найденных проблемах. В следующих главах вы увидите, как на практике происходит раннее тестирование.

В целом этот подход воплощает в себе девиз: «Качество — это ответственность команды», потому что проверки качества на каждом этапе жизненного цикла разработки программного обеспечения — прототипов приложений, требований и т. д., как говорилось ранее, — должны проводить разные члены команды. Таким образом, можно утверждать, что развитие навыков выполнения различных проверок качества у всех членов команды имеет решающее значение для создания высококачественного ПО!

¹ *Бек К. Экстремальное программирование: разработка через тестирование.* — СПб.: Питер, 2024.

Десять навыков фулстек-тестирования

Размышляя о навыках тестирования, мы склонны объединять их в две широкие категории — ручное и автоматизированное тестирование. Но за последние несколько десятилетий технологии развились, и в эти широкие категории были включены новые важные навыки, которым необходимо обучаться для выполнения различных проверок качества и создания высококачественных мобильных веб-приложений. На рис. 1.4 приведены десять навыков фулстек-тестирования, которые позволяют эффективно выполнять его.

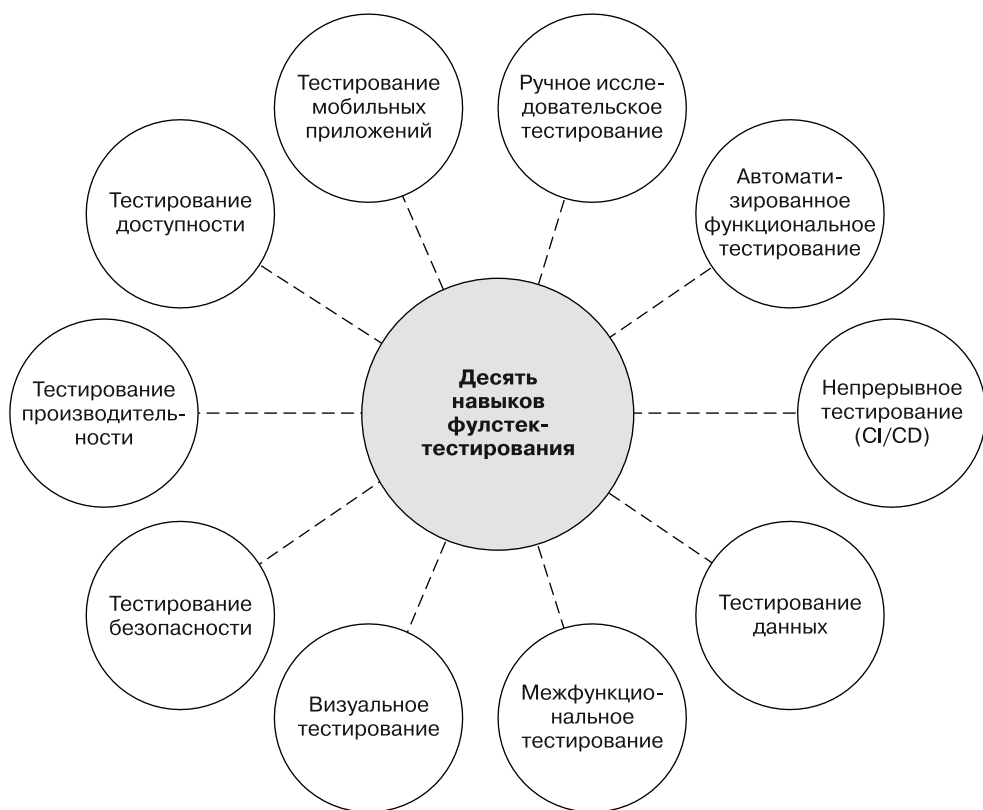


Рис. 1.4. Десять навыков фулстек-тестирования, владеть которыми обязательно для создания высококачественных мобильных веб-приложений

Рассмотрим эти десять навыков и узнаем, почему их стоит освоить.

Ручное исследовательское тестирование

Для начала уточним: ручное исследовательское тестирование отличается от ручного тестирования. Последнее относится к проверке заданного списка требо-

ваний и не всегда требует аналитического мышления. Ручное исследовательское тестирование, напротив, — это умение вникать в детали приложения, придумывать различные сценарии применения, помимо тех, что задокументированы в пользовательских историях, моделировать их в тестовой среде и наблюдать за поведением приложения. Этот навык требует логического и аналитического мышления и является первым и главным навыком, необходимым для создания качественных приложений. Существуют разные методологии и подходы к структурированию исследовательских сессий. Мы обсудим их в главе 2.

Автоматизированное функциональное тестирование

Это один из основных навыков, необходимых для раннего тестирования. Автоматизация тестирования значительно сокращает ручной труд, что особенно важно, когда приложение расширяется и в него включается все больше функций. Проще говоря, этот навык заключается в написании кода для автоматизированного тестирования требований к функциям, протекающего без вмешательства человека. Для этого нужны инструменты, поэтому для приобретения данного навыка требуется знание различных инструментов, которые можно использовать при написании тестов для разных уровней приложения. Однако на этом дело не заканчивается — необходимо также знать, какие антипаттерны следует искать при автоматизированном тестировании и как их избегать. Различные аспекты этого навыка обсудим в главе 3.

Непрерывное тестирование

Непрерывная доставка — это практика постепенной доставки функций конечным пользователям в ходе коротких циклов, а не в рамках одного масштабного выпуска. Благодаря непрерывной доставке компания начинает получать прибыль уже на раннем этапе и может быстро оценить и пересмотреть свою стратегию на основе отзывов конечных пользователей. Для непрерывной доставки необходимо постоянно тестировать приложение, чтобы оно всегда было готово к выпуску. Самый простой способ добиться этого — автоматизировать и интегрировать проверки качества в конвейеры CI/CD и запускать их как можно чаще, чтобы упростить тестирование. Навыки непрерывного тестирования включают в себя определение типов автоматических тестов, которые следует запускать на каждом этапе цикла доставки, чтобы команда могла эффективно интегрировать их в конвейеры CI/CD и быстрее получать обратную связь. Об этом подробно рассказывается в главе 4.

Тестирование данных

Возможно, вы слышали такие выражения: «Данные — это деньги» и «Данные — это новая нефть». Они подчеркивают, насколько в настоящее время важно тестировать целостность данных. Когда приложение теряет или показывает неверные данные, пользователи утрачивают доверие к нему. Навыки тестирования данных требуют знания различных типов систем хранения и обработки данных, обычно применяемых в мобильных и веб-приложениях (базы данных, кэши, потоки событий и т. д.), а также умения создавать подходящие тестовые

сценарии. Подробнее об этих темах, а также о том, как на основе потоков данных между компонентами приложения создавать новые тестовые сценарии, мы поговорим в главе 5.

Визуальное тестирование

Внешний вид приложения — один из основных факторов, влияющих на ценность бренда, особенно когда речь идет о крупных продуктах, реализующих взаимодействие бизнеса с потребителями и применяемых миллионами. Непривлекательный внешний вид может отрицательно повлиять на ценность бренда. Поэтому важно убедиться, что конечные пользователи получают гармоничный и приятный визуальный опыт, и для этого проводить визуальное тестирование приложения. Для эффективного визуального тестирования важно понимать, как компоненты пользовательского интерфейса взаимодействуют друг с другом и с браузером (в случае веб-приложений). Тестирование этого вида тоже можно автоматизировать с помощью инструментов, отличных от тех, что применяются для автоматизации функционального тестирования. Об этом навыке и различиях между этими двумя типами автоматизации мы поговорим в главе 6.

Тестирование безопасности

Нарушения в системе защиты давно стали притчей в современном мире. И даже такие гиганты, как Facebook и Twitter, не являются исключениями. Проблемы защиты дорого обходятся и конечным пользователям, и бизнесу с точки зрения потери или раскрытия конфиденциальной информации, юридических санкций и репутации бренда. Прежде тестирование безопасности считалось узкоспециализированным навыком, и квалифицированных тестировщиков этого профиля обычно привлекали лишь к концу цикла разработки, чтобы с их помощью отыскать проблемы безопасности. Но из-за нехватки профессионалов в сфере тестирования безопасности и растущего числа случаев взлома защиты командам разработчиков программного обеспечения рекомендуется включать базовое тестирование безопасности в повседневную работу. В главе 7 мы поговорим о том, как научиться мыслить подобно хакеру и выявлять проблемы безопасности в приложениях, а также обсудим инструменты для автоматизации сканирования системы защиты.

Тестирование производительности

Даже незначительное снижение производительности может привести к огромным финансовым и репутационным потерям — вспомните пример Flipkart. В число навыков тестирования производительности входит умение измерять набор ключевых показателей производительности на разных уровнях приложения. Тесты производительности тоже можно автоматизировать и интегрировать в конвейеры CI для получения постоянной обратной связи. Стратегию раннего тестирования производительности и соответствующие инструменты мы обсудим в главе 8.

Тестирование доступности

Мобильные и веб-приложения стали товаром повседневного потребления. А их доступность для людей с ограниченными возможностями не только требуется законодательствами многих стран, но и является правильной с этической точки зрения. Чтобы приобрести навыки тестирования доступности, нужно сначала понять стандарты доступности, устанавливаемые законом, а уже затем вручную или с помощью инструментов автоматического аудита доступности проверить, соблюдаются ли они. Этот навык, а также причины того, что обеспечение доступности может даже приносить прибыль, мы обсудим в главе 9.

Межфункциональное тестирование

Как мы видели, конечные пользователи и предприятия имеют длинный список требований к качеству, таких как доступность, масштабируемость, удобство обслуживания, наблюдаемость и т. д., помимо безошибочной работы. Они называются *межфункциональными требованиями* приложения. Функциональные требования, как правило, привлекают наибольшее внимание, но именно межфункциональные требования обеспечивают приложению особое качество, и отказ от межфункционального тестирования может привести к неудовлетворенности бизнеса, команд разработчиков, конечных пользователей или всех сразу. Поэтому навыки межфункционального тестирования считаются одними из основных. Подробнее приемы и инструменты для межфункционального тестирования мы обсудим в главе 10.



Многие также называют межфункциональные требования нефункциональными требованиями. О тонкостях различия этих двух терминов поговорим в главе 10.

Тестирование мобильных приложений

Огромное количество приложений, которые были доступны в 2021 году в ведущих магазинах приложений (Google Play и Apple App Store), может показаться ошеломляющим — 5,7 млн (<https://oreil.ly/L47MG>). Взрывной рост числа мобильных приложений в основном обусловлен распространением мобильных устройств. Как заявила компания веб-аналитики Global Stats в 2016 году, количество мобильных устройств, подключенных к Интернету, превысило количество настольных компьютеров (<https://oreil.ly/mL3YF>). По этой причине умение тестировать мобильные приложения и совместимость веб-сайтов с мобильными устройствами сейчас считается критически важным навыком.

Для тестирования мобильных приложений необходимы все перечисленные ранее навыки, а вдобавок — изменение мышления. Кроме того, следует изучить целый набор инструментов, специфичных для мобильных устройств, чтобы тестировать различные параметры качества мобильных приложений. Поэтому мобильное тестирование в этой книге выделено в отдельный навык. Нюансы мобильной среды мы рассмотрим в главе 11.

Все вместе эти десять навыков фулстек-тестирования позволят вам протестировать весь спектр аспектов качества мобильных и веб-приложений. Как упоминалось ранее, каждому члену команды важно приобрести определенный уровень компетентности в каждом из этих навыков. Книга расскажет вам, как этого добиться, и покажет множество практических примеров.

Ключевые выводы

- Качество программного обеспечения больше нельзя приравнивать к простому функционированию без ошибок. Приложение не может считаться качественным, если его параметры качества (безопасность, производительность, визуальная привлекательность и т. д.) не находятся на одном уровне.
- Фулстек-тестирование предполагает тестирование всех показателей качества приложения на каждом уровне, что позволяет получить высококачественное программное обеспечение.
- Чтобы фулстек-тестирование достигло своей главной цели — обеспечило доставку высококачественного ПО, командам следует приступать к тестированию на ранних этапах, начинать его одновременно с анализом и выполнять на протяжении всего цикла разработки и поставки.
- Раннее тестирование воплощает афоризм: «Качество — ответственность команды», потому что требует, чтобы каждый член команды взял на себя ответственность за выполнение определенных проверок качества на разных этапах разработки и поставки. Это требует повышения квалификации и приобретения соответствующих навыков тестирования на различных уровнях компетентности.
- Две классические категории навыков тестирования — ручное и автоматизированное — были расширены и в настоящее время включают в себя целый набор новых навыков, необходимых для эффективного фулстек-тестирования. В этой главе были перечислены десять навыков тестирования, необходимых для создания современных высококачественных мобильных и веб-приложений, которые мы рассмотрим в следующих главах.

Ручное исследовательское тестирование

Не все, кто блуждает, заблудились.

Дж. Р. Р. Толкин

Ручное исследовательское тестирование — довольно трудоемкое занятие, в ходе которого вы тестируете приложение с целью изучить и понять его поведение в различных ситуациях, которые нигде явно не сформулированы — ни в документе с требованиями, ни в пользовательских историях, ни где-либо еще. В результате исследования часто обнаруживаются новые последовательности действий пользователя, не предусмотренные на этапе анализа или разработки, а также ошибки в существующих последовательностях действий. Открытие таких несоответствий приносит необычайную радость нашедшему их, потому что демонстрирует его аналитические способности и наблюдательность!

Обычно ручное исследовательское тестирование проводится в тестовой среде, где приложение разворачивается целиком. Тестировщики берут на себя смелость вмешиваться в различные компоненты приложения, такие как база данных, сервисы или фоновые процессы, чтобы смоделировать различные сценарии и понаблюдать за поведением приложения. Этим исследовательское тестирование отличается от традиционного ручного тестирования, суть которого заключается в выполнении вручную определенного набора действий, описанного как критерии приемки в пользовательских историях или в документе с требованиями, и проверки соответствия заявленным ожиданиям. Другими словами, ручное тестирование не всегда требует аналитических навыков, тогда как исследовательское тестирование открывает перед тестировщиками широкий простор, предлагая им выйти за рамки задокументированного и даже за пределы того, что на данный момент известно о приложении!

Учитывая некоторое совпадение ручного и исследовательского тестирования, часть команд недооценивают важность последнего. Также бытует мнение, что объема анализа, проведенного в рамках разработки и обсуждения пользовательской истории, более чем достаточно, особенно если он дополнен автоматизированным тестированием (автоматизированное тестирование подробно обсуждается в главе 3). Однако

при этом упускается из виду тот факт, что во время обсуждения пользовательской истории анализ обычно выполняется с точки зрения бизнеса, а во время разработки программисты могут сосредоточиться на текущем объеме функциональности и думать только о данном небольшом фрагменте. Как следствие возникает очевидный пробел — приложение не исследуется с точки зрения конечного пользователя и не рассматривается с точки зрения общей картины в среде развертывания. Из-за этого могут остаться незамеченными проблемы интеграции и пропуски в последовательностях действий конечных пользователей. Именно поэтому необходимо проводить ручное исследовательское тестирование после разработки.



Исследовательское тестирование объединяет все три аспекта — требования бизнеса, детали технической реализации и потребности конечного пользователя — и проверяет все, что считается истинным, со всех этих точек зрения. Хорошей практикой считается признавать функциональность реализованной до конца только после того, как будут созданы тестовые сценарии и автоматизировано тестирование новых последовательностей действий пользователей, обнаруженных в ходе исследовательского тестирования.

Возможно, нет необходимости назначать отдельного специалиста для проведения исследовательского тестирования после разработки, хотя этот подход может оказаться наилучшим, поскольку этот человек приобретает богатый практический опыт и оттачивает навыки наблюдения и анализа. Если этому препятствуют соображения стоимости или доступности, то существующие члены команды должны взять на себя ответственность за проведение исследовательского тестирования во время каждой итерации по циклическому принципу. Действительно, развитие навыков исследовательского тестирования может помочь каждому члену команды лучше справляться со своими обязанностями.

Если вы один из членов команды и хотите развить у себя навыки исследовательского тестирования, то эта глава для вас. Далее мы обсудим хорошо зарекомендовавшие себя подходы, которые могут помочь в исследовательском тестировании, а также стратегию решения этой задачи. Упражнения, приводимые в данной главе, сосредоточены, в частности, на исследовательском тестировании веб-интерфейсов и API. Мы также рассмотрим ряд полезных методов поддержания гигиены тестовой среды, поскольку полноценная тестовая среда играет ключевую роль в успехе ручного исследовательского тестирования.

ЧАСТО ИСПОЛЬЗУЕМЫЕ ТЕРМИНЫ

Далее перечислены некоторые часто применяемые термины, которые вы встретите в этой главе.

- *Функция или функциональность* — определяет ценность, предоставляемую приложением своим конечным пользователям. Например, вход в систему — это функциональность, обеспечивающая безопасность конечных пользователей.

- *Пользовательский сценарий* — набор действий, которые конечный пользователь выполняет в приложении для достижения цели, предоставленной функциональностью. Например, чтобы войти в систему, он должен ввести свои учетные данные и войти в систему — это пользовательский сценарий.
- *Тестовый сценарий* — набор действий, проверяющих работу функциональности. Например, ввод валидного имени пользователя и пароля и проверка успешного входа в систему — это тестовый сценарий. Аналогично ввод невалидного имени пользователя и проверка наличия сообщения об ошибке — тоже тестовый сценарий. Первый является позитивным тестовым сценарием, поскольку позволяет успешно достичь цели, предоставляемой функциональностью, а второй — негативным тестовым сценарием, так как не позволяет достичь цели. Для полного изучения функциональности необходимо смоделировать и проверить как позитивные, так и негативные тестовые сценарии.
- *Граничный (или крайний) случай* — негативный тестовый сценарий, который встречается очень редко.

Введение

Для начала рассмотрим восемь подходов к исследовательскому тестированию и практические примеры их использования, а затем попрактикуемся в изучении функциональности.

Подходы к исследовательскому тестированию

Цель обсуждения подходов к исследовательскому тестированию — помочь сформировать мысленные модели, которые можно интуитивно применять к соответствующим контекстам приложения. Главная задача подходов — сузить объем тестирования, придав ясность выбранной части функциональности и структурировав ее. Например, в приложениях часто встречаются поля для ввода чисел. Вместо случайного опробования всех возможных числовых значений для проверки такого поля предлагается логически разделить входные данные на наборы выборок. Существуют также подходы, помогающие структурировать бизнес-правила и увидеть различные пользовательские и тестовые сценарии. Рассмотрим их на примерах по очереди.

В качестве первого примера возьмем веб-страницу, которая просит пользователя ввести сумму дохода (рис. 2.1) и отображает рассчитанную сумму налога. Справа показаны различные налоговые группы, применяемые для расчета.

Чтобы проверить работу логики вычисления налога, нужно определить позитивные и негативные тестовые сценарии, чтобы потом использовать их в качестве входных данных. Здесь следует отметить, что доход — это числовое значение в диапазоне от 0 до бесконечности. Чтобы получить позитивные и негативные тестовые сценарии, нужно логически сузить набор правильных входных значений

и проверить получаемые результаты. Помочь в этом могут два подхода: разделение на классы эквивалентности и анализ граничных значений.

Проверьте свой подоходный налог!

Введите
общий доход

Отправить

Подробную информацию о расчете
налога см. в таблице справа

Расчет налогообложения

Доход	Налог
\$0–5000	5 %
\$5000–15000	10 %
>\$15000	30 %

Рис. 2.1. Простой калькулятор налогов

Разделение на классы эквивалентности

Этот прием предполагает деление входных данных на группы, элементы которых приводят к одному и тому же результату или подвергаются аналогичной обработке, после чего достаточно выбрать по одному образцу из каждого класса, чтобы полностью протестировать функциональность.

Применив этот подход к примеру с налоговым калькулятором, выделим три класса налогообложения: [0–5000], [5001–15 000] и [>15000]. Эти три класса можно считать *эквивалентными*, поскольку все входные данные внутри каждого из них будут обрабатываться с применением одних и тех же правил. Соответственно, для проверки позитивных тестовых сценариев достаточно протестировать три точки входных данных, по одной из каждого класса. Например, если вы заняты и не хотите опробовать больше примеров, то достаточно проверить результаты для входных данных 2000, 10 000 и 20 000, чтобы подтвердить позитивные тестовые сценарии. Затем то же самое можно сделать для проверки негативных тестовых сценариев. Вот, например, классы входных данных, которые должны привести к ошибке: [отрицательные значения], [буквы], [прочие символы] и т. д. Опять же достаточно проверить по одному значению из каждого класса.

Этот подход может пригодиться и для модульного тестирования (обсуждается в главе 3), и для проверки любого другого контекста приложения, например для тестирования результатов, основанных на времени (до и после события), внутренних состояний системы и т. д.

Анализ граничных значений

Анализ граничных значений дополняет подход с разделением на классы эквивалентности явной проверкой граничных условий в каждом из классов. Это может пригодиться при поиске ошибок, потому что граничные условия часто бывают нечетко определены и поэтому реализуются неправильно. В примере с налоговым калькулятором требования к налоговым категориям могли быть сформулированы так: «С дохода меньше 5000 долларов взимается 5 % налога, с дохода от 5000 до 15 000 долларов — 10 % и с дохода более 15 000 долларов — 15 %». Такое описание недостаточно четко определяет граничные условия, то есть в какие классы должны попадать значения 5000 и 15 000. Анализ граничных значений позволяет привлечь внимание к таким проблемам и помогает решить их, проверяя граничные значения в каждом из классов эквивалентности в дополнение к проверке данных внутри класса.

Применим этот подход к примеру с налоговым калькулятором и проанализируем граничные значения в каждом из классов, выявленных ранее. Класс [0–5000] имеет граничные значения 0 и 5000. Но по логике вещей с нулевого дохода налог взиматься не должен. Поэтому сформируем новые классы эквивалентности: [0] и [1–5000]. Таким образом, чтобы охватить все позитивные тестовые сценарии, мы должны проверить следующие граничные значения: [0, 1, 5000, 5001, 15 000, 15 001], как показано на рис. 2.2.



Рис. 2.2. Классы эквивалентности с граничными условиями

Как показывает этот пример, рассматриваемый подход принесет максимальную пользу, если станет применяться на всех этапах поставки, начиная с анализа.

Обсудив два подхода, помогающие структурировать входные значения для одного поля и преобразовать их в минимальный набор позитивных и негативных тестовых сценариев, перейдем к немного более сложным сценариям, когда разные комбинации входных данных дают разные результаты. Возьмем за основу классический пример страницы входа, которая принимает два входных значения — адрес электронной почты и пароль, и обсудим, как диаграмма переходов и состояний, таблица решений и схема причинно-следственных связей могут помочь визуализировать различные тестовые сценарии.

Диаграмма переходов и состояний

Диаграмма переходов и состояний может пригодиться при разработке тестовых сценариев, когда поведение приложения меняется в зависимости от истории входных данных. Например, на странице входа может отображаться сообщение об ошибке после первой и второй попыток ввести неправильный пароль, а после третьей такой попытки учетная запись может блокироваться. В таких сценариях, чтобы получить тестовые сценарии, желательно нарисовать дерево переходов, как показано на рис. 2.3. В дереве переходов каждое состояние приложения изображается как узел. Возможные результаты действий показаны в виде подузлов, а сами действия/события, ведущие к тем или иным результатам, — как метки ветвей.

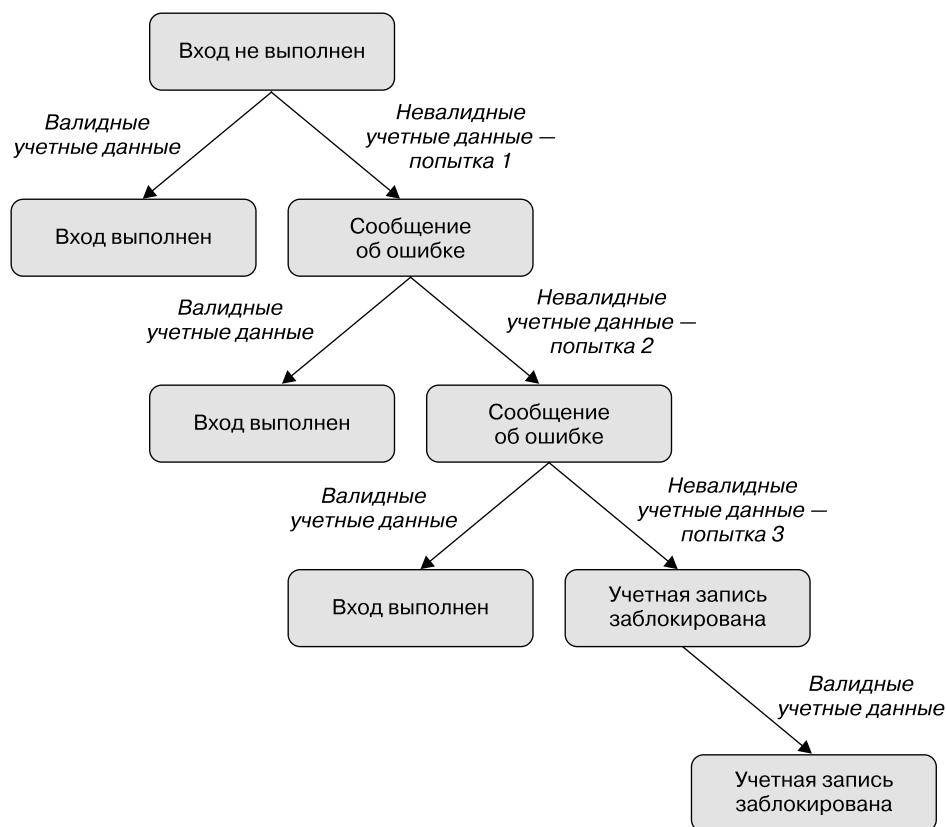


Рис. 2.3. Дерево переходов между состояниями в сценарии ошибочного ввода пароля

Это дерево дает четкое представление о каждом тестовом сценарии: начальном состоянии, выполняемом действии, изменяющем состояние приложения, и ожидаемых результатах, которые следует проверить. Построение диаграммы с деревом

также помогает оценить объем усилий, необходимых для тестирования функции, точно отражая количество состояний и переходов, что может помочь на этапе планирования.

Переходы между состояниями могут быть гораздо более сложными, например, в системе управления заказами, где последние проходят через такие состояния, как подтверждение оплаты, ожидание, отгрузка, отмена, вручение и т. д. В таких случаях визуализация состояний в виде узлов и действий, приводящих к смене состояния, даст четкое представление о самой функции.

Таблица решений

Когда для получения результатов входные данные логически объединяются (по И, ИЛИ и т. д.), для построения тестовых сценариев можно использовать таблицы решений. Этот подход может сэкономить много времени на этапе тестирования, потому что все возможные входные комбинации и ожидаемые результаты заранее четко обозначены в таблице. В примере с входом в систему адрес электронной почты и пароль логически связаны по И, то есть для успешного входа в систему правильными должны быть оба входных значения: адрес электронной почты И пароль. В табл. 2.1 показана таблица решений для этого сценария.

Этот прием может сэкономить время, позволяя исключить некоторые ненужные тестовые сценарии. Например, в сценарии входа в систему тестовый сценарий 3, в котором оба входных значения неверны, можно исключить, потому что вход в систему невозможен, если хотя бы одно из входных значений неверно.

Таблица 2.1. Таблица решений для сценария входа

Таблица решений		Тестовый сценарий 1	Тестовый сценарий 2	Тестовый сценарий 3	Тестовый сценарий 4
Условия	Адрес	True	False	False	True
	Пароль	False	True	False	True
Действия	Вход	—	—	—	True
	Сообщение об ошибке	Да	Да	Да	—

Диаграмма причинно-следственных связей

Еще один способ визуализации логически связанных входных данных и возможных результатов — это диаграмма причинно-следственных связей. Данный подход помогает увидеть общую картину и, следовательно, особенно полезен на этапе анализа. Нарисовав диаграмму, вы сможете преобразовать ее в таблицу решений и получить подробные тестовые сценарии. На рис. 2.4 показана диаграмма причинно-следственных связей для нашего примера входа в систему.

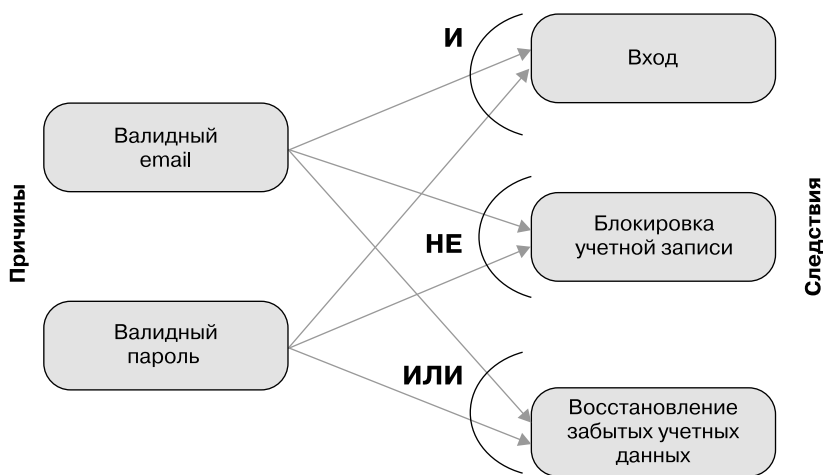


Рис. 2.4. Диаграмма причинно-следственных связей для сценария входа

С одной стороны здесь перечислены причины, с другой — следствия, а между ними с помощью логических операторов задаются связи.

Подходы, которые мы рассмотрели, помогают структурировать входные данные, связанные друг с другом. А далее изучим два подхода, которые помогают работать с множествами независимых переменных и большими наборами данных.

Попарное тестирование

В приложениях часто приходится иметь дело с несколькими входными значениями, и порой бывает сложно управлять их вариациями и получать тестовые сценарии. Попарное тестирование, также известное как тестирование всех парных комбинаций, помогает свести к минимуму количество тестовых сценариев, когда результаты зависят от нескольких таких независимых переменных/входных данных. Приведу короткое упражнение, чтобы понять суть этого подхода.

Рассмотрим форму, в которой вводятся три независимых значения: тип операционной системы (ОС), производитель устройства и разрешение экрана. Поле ввода типа ОС может принимать два значения: Android или Windows. Поле ввода производителя устройства может принимать три значения: Samsung, Google или Oppo. Наконец, поле ввода разрешения экрана может принимать значения «Низкое», «Среднее» и «Высокое». Итак, при тестировании этой формы мы можем получить $2 \times 3 \times 3 = 18$ комбинаций входных значений (табл. 2.2).

Попарное тестирование предполагает, что любую пару входных данных достаточно проверить один раз, потому что они являются независимыми переменными. В результате список тестовых сценариев сокращается до девяти (табл. 2.3).

Таблица 2.2. Примеры тестовых сценариев без применения метода попарного тестирования

Тестовый сценарий	Производитель	Разрешение	ОС
1	Samsung	Низкое	Android
2	Samsung	Среднее	
3	Samsung	Высокое	
4	Google	Низкое	
5	Google	Среднее	
6	Google	Высокое	
7	Oppo	Низкое	
8	Oppo	Среднее	
9	Oppo	Высокое	
10	Samsung	Низкое	Windows
11	Samsung	Среднее	
12	Samsung	Высокое	
13	Google	Низкое	
14	Google	Среднее	
15	Google	Высокое	
16	Oppo	Низкое	
17	Oppo	Среднее	
18	Oppo	Высокое	

Таблица 2.3. Сокращенный список тестовых сценариев после применения метода попарного тестирования

Тестовый сценарий	Производитель	Разрешение	ОС
1	Oppo	Низкое	Android
2	Samsung	Низкое	Windows
3	Google	Низкое	Android
4	Oppo	Среднее	Windows
5	Samsung	Среднее	Android
6	Google	Среднее	Windows
7	Oppo	Высокое	Android
8	Samsung	Высокое	Windows
9	Google	Высокое	Android/Windows

Из прежнего списка были исключены несколько повторяющихся пар. Например, пары [Google, Среднее] и [Google, Windows] теперь встречаются только один раз.

Выборка

До сих пор мы имели дело с небольшим количеством входных данных, перечислить которые сможет любой из нас без помощи инструментов. Но как быть, если потребуется протестировать большие наборы данных? Представьте, что мы написали новую систему страхования для замены старой и нам нужно проверить, правильно ли были перенесены существующие сведения о страховании. В старой системе могли храниться данные миллионов пользователей, поэтому нереально задействовать обсуждавшиеся до сих пор подходы для получения тестовых сценариев. Например, мы не можем определить классы эквивалентности, потому что каждый пользователь будет характеризоваться уникальной комбинацией значений, определяющих возраст, способ уплаты страховых взносов, продолжительность страхования, вид страхования и т. д., и не можем применить попарное тестирование, так как количество переменных слишком велико, чтобы выявить и устранить повторяющиеся пары. В таких случаях прибегают к методу выборки.

Обычно этот прием можно применять к любым непрерывным входным данным большого объема. Он включает в себя выбор подмножества значений для тестирования (рис. 2.5), обычно с использованием одного из следующих методов: случайная выборка или выборка по конкретным критериям.

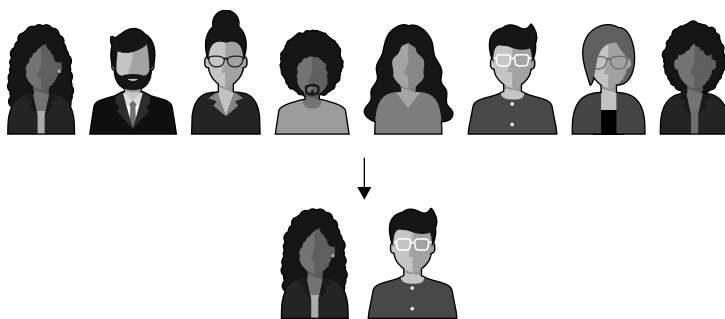


Рис. 2.5. Случайная выборка или выборка по конкретным критериям

Применяя случайную выборку, мы произвольно выбираем случайные образцы из набора данных и проверяем результаты, получаемые с их участием. Например, если имеется 1000 пользователей, мы можем случайным образом выбрать 50–100 из устаревшей системы и сравнить эти данные с данными, хранящимися в новой системе. Выполняя выборку по конкретным критериям, мы выбираем об-

разцы по каким-то общим характеристикам. Например, для тестирования системы страхования могли бы выбрать образцы по критериям, характеризующим пользователей, таким как возраст, продолжительность страхования (количество лет подписки), способ уплаты страховых взносов, профессия и т. д., или по критериям, характерным для страхового полиса, таким как периодичность платежей, цена и т. д. Мы могли бы усовершенствовать эту технику, задав количество образцов, выбираемых по каждому критерию, пропорциональным фактическому распределению значений в наборе данных. Это даст репрезентативный мини-набор данных и с высокой вероятностью охватит все виды тестовых сценариев.

И наконец, рассмотрим последний подход, который связан скорее с тренировкой навыков аналитического и логического мышления, чем с набором фиксированных рекомендаций.

Метод угадывания ошибок

Угадывание ошибок предполагает прогнозирование возможных проблем на основе прошлого опыта. К ним могут относиться общие проблемы с интеграцией, проверкой входных данных, граничными случаями и т. д. Предыдущий опыт играет решающую роль в прогнозировании возможных ошибок, кроме того, можно использовать свое понимание технологии и логические рассуждения. Развитие такого рода мышления улучшает навыки исследовательского тестирования по всем направлениям.

Вот несколько типов ошибок, которые, как показывает мой опыт, возникают регулярно.

- Отсутствие проверки недопустимых/пустых входных значений и отсутствие соответствующих сообщений об ошибках, предлагающих пользователю исправить вводимые данные.
- Непонятные коды состояния HTTP, возвращаемые механизмом проверки данных, технические и бизнес-ошибки (некоторые из них рассмотрим в подразделе «Тестирование API» далее в этой главе).
- Отсутствие проверки граничных условий, характерных для предметной области, типов данных, состояний и т. д.
- Отсутствие проверки на стороне пользовательского интерфейса технических ошибок, таких как сбой сервера, истечение времени ожидания ответа и т. д.
- Проблемы пользовательского интерфейса (например, искажения и артефакты), возникающие во время переходов, обновления данных и навигации.
- Использование ключевых слов SQL, например ключевого слова `like` и оператора «равно», как взаимозаменяемых, из-за чего могут полностью меняться результаты.
- Неочищенные кэши и неопределенные тайм-ауты сеансов.

- Повторная отправка запроса, когда пользователь нажимает кнопку Назад в браузере.
- Отсутствие проверки формата файла при его загрузке с разных платформ.

Вы можете задействовать эти восемь подходов к исследовательскому тестированию, чтобы сосредоточиться на изучении функциональности и создании хороших тестовых сценариев. Обратите внимание на то, что эти подходы можно применять к любым аспектам приложений, не только к входным данным. Теперь, познакомившись с этими подходами, рассмотрим подробнее, что входит в изучение функциональности, а затем перейдем к практическим упражнениям.

Изучение функциональности

Предположим, вас попросили провести исследовательское тестирование функции создания заказов в приложении электронной коммерции. С чего вы начнете? Этот раздел ответит на данный вопрос и покажет четыре основных пути изучения любого конкретного приложения (рис. 2.6).



Рис. 2.6. Четыре основных пути изучения функциональности

Функциональные пользовательские сценарии

Под функциональными пользовательскими сценариями в приложении понимаются последовательности действий, которые совершает конечный пользователь, работая с приложением, например: вход в систему, поиск продукта, добавление его в корзину, указание адреса доставки, выбор варианта доставки, оплата и, наконец, получение подтверждения о приеме заказа. Это так называемый позитивный однопользовательский сценарий, и именно его следует проверить в первую очередь. Для этого нужно изучить позитивный сценарий с различными адресами доставки, способами оплаты и доставки, а также комбинациями товаров, чтобы убедиться, что он работает безотказно.

В ходе исследования вы можете обнаружить, что применяете некоторые из подходов исследовательского тестирования, обсуждавшихся ранее. Например, вы можете использовать методы разделения на классы эквивалентности и анализа граничных значений, чтобы проверить, добавляет ли приложение правильную сумму налога к общей цене. Можно задействовать также дерево переходов для получения тестовых сценариев с разными адресами и методами доставки, доступными для этой комбинации адресов. Как только вы убедитесь, что позитивный однопользовательский сценарий работает безупречно, начинайте изучать пути двух других типов.

Повторяющиеся сценарии

Конечные пользователи часто повторяют один и тот же процесс (или его части) по несколько раз, например, могут отыскивать разные продукты и добавлять их в корзину. Но обычно пользовательский сценарий тестируется только один раз, и если он работает, то повторные попытки пройти его не тестируются, потому что предполагается, что от повторения поведение приложения не изменится. Однако на практике так бывает не всегда. Например, если пользователь попытается еще раз добавить тот же товар в корзину, то пользовательский интерфейс может сообщить, что он уже добавлен, и предложить увеличить количество его единиц. Повторение сценариев тоже следует тестировать.

Многопользовательские сценарии

Функция может работать идеально, когда ею пользуется один человек, но вести себя неожиданно при обслуживании сразу нескольких. Поэтому важно изучить возможные коллизии, когда действия одного пользователя влияют на другого. Например, что произойдет, если два разных человека одновременно добавят в свои корзины последнюю доступную единицу товара?

Функциональные пользовательские сценарии часто выбираются первыми для исследования при изучении приложения, но внутри сценариев может быть несколько ответвлений, которые тоже нужно внимательно изучить. К наиболее важным ветвям относятся позитивные однопользовательские сценарии, повторяющиеся сценарии и многопользовательские сценарии.

Обработка сбоев и ошибок

Как отмечалось в начале главы, исследовательское тестирование проводится в тестовой среде и включает в себя вмешательство в компоненты приложения для моделирования различных реальных сценариев и наблюдения за поведением приложения. В этом утверждении есть два словосочетания, которые требуют особого внимания, поскольку составляют основу исследовательского тестирования: *вмешательство в компоненты приложения* и *сценарии реального времени*. При рассмотрении сценариев реального времени следует также учитывать все возможные сбои, потому что они практически неизбежны. Например, в ходе взаимодействий с компонентами приложения может произойти сбой сети, не позволяющий отправить ответ пользователю, или сеть может работать медленно и вызывать задержки, или из-за аппаратного сбоя сервисы приложения могут оказаться недоступными. Все эти и другие сбои необходимо предвидеть и смоделировать в тестовой среде во время исследовательского тестирования.

Помимо вышеупомянутых сбоев сети, сервисов и оборудования, могут возникать ошибки из-за недопустимых действий пользователя. Разработку функциональности можно считать завершенной, только если она имеет встроенные проверки и обрабатывает все такие случаи. В функции создания заказов есть несколько пунктов, требующих изучения подобных проверок. Например, как обсуждалось ранее, страница входа должна проверять адреса электронной почты и пароли, текст строки поиска следует проверять на наличие недопустимых входных данных и доступность искомого и т. п. Также следует проверить адрес доставки и платежные реквизиты, наличие товаров в корзине и т. д.

Исследовательское тестирование должно уделять большое внимание выявлению возможных сбоев и обработке ошибок. Функция должна информировать пользователей об их ошибках и предлагать возможные пути решения, предоставляя осмысленный текст сообщения об ошибке.

Внешний вид пользовательского интерфейса

Пользовательский интерфейс — это то, что видит конечный пользователь, и с его качеством не может быть явных проблем. Поэтому его внешний вид и ощущения, испытываемые в ходе работы с ним, — еще один важный аспект, который стоит изучить. Приведу лишь несколько примеров: тестирование качества пользовательского интерфейса для функции создания заказов может включать проверку достаточности места для отображения адреса доставки (места должно быть достаточно, чтобы отобразить длинное название улицы, но не слишком много, чтобы не оставалось большое пустое пространство, когда адрес короткий) и качества изображений товаров. Конечные пользователи должны иметь возможность беспрепятственно управлять приложением из предпочитаемых ими браузеров, а при выполнении продолжительных операций должен отображаться значок загрузки. Структурированный подход к тестированию качества пользовательского интерфейса подробно обсуждается в главе 6.

Межфункциональные аспекты

В любой конкретной функциональности может иметься несколько межфункциональных аспектов, таких как безопасность, производительность, доступность, аутентификация, авторизация, возможность аудита, конфиденциальность и т. д., которым следует уделять особое внимание в ходе исследовательского тестирования. Некоторым из этих аспектов из-за их важности посвящены отдельные главы книги. Вот несколько межфункциональных требований, которые желательно исследовать при изучении функциональности создания заказов.

Безопасность

В процессе создания заказа злоумышленник может попытаться ввести SQL-запросы в поля ввода пользовательского интерфейса и взломать приложение. В нем должны быть предусмотрены проверки для противостояния этим действиям. Аналогично данные кредитных карт пользователей не должны храниться в базе данных в открытом виде и не должны регистрироваться в логах приложения, чтобы исключить их утечку в случае взлома. Все эти аспекты тестирования безопасности и многое другое подробно обсуждается в главе 7.

Конфиденциальность

Личные данные пользователей, такие как номера кредитных карт и адреса доставки, не должны сохраняться в базе данных приложения без их согласия. Кроме того, люди должны быть заранее проинформированы о том, как их данные могут применяться для аналитики и будут ли они передаваться сторонним сервисам для обработки. Некоторые аспекты конфиденциальности данных регулируются правовыми нормами, мы обсудим эти вопросы в главе 10.

Аутентификация/авторизация

Большинство веб-сайтов имеют функцию аутентификации пользователей, которая требует изучения с применением тестовых сценариев и проверки таких возможностей, как единый вход, двухфакторная аутентификация, истечение срока сеанса, блокировка и разблокировка учетной записи и т. д. Приложение электронной коммерции может разрешить конечным пользователям просматривать каталог товаров без авторизации, но не оформлять заказ.

Аналогично могут быть предусмотрены роли (например, администратора, исполнительного директора по работе с клиентами) и разрешения (например, редактирование заказа), назначенные разным пользователям, что требует изучения тестовых сценариев, связанных с авторизацией, таких как назначение нескольких ролей с перекрывающимися привилегиями, добавление новых разрешений в существующие роли, наблюдение за поведением приложения при выполнении операций без необходимых разрешений и т. д.

Это лишь несколько примеров. Около 30 межфункциональных аспектов и способы их тестирования обсуждаются в главе 10.

Описанные здесь четыре направления исследований должны привести к всестороннему тестированию любой заданной функциональности. Обратите внимание: при исследовании каждого из этих направлений могут появиться новые идеи и тестовые сценарии, не соответствующие им. Важно записать их, чтобы вернуться к ним позже или использовать для исследования других направлений.

Стратегия ручного исследовательского тестирования

Стратегия ручного исследовательского тестирования (рис. 2.7) объединяет все, что обсуждалось до сих пор, а также командные процессы. Она определяет практическую схему исследовательского тестирования в повседневной работе над проектом. Начнем с внешнего полукруга и будем двигаться внутрь.

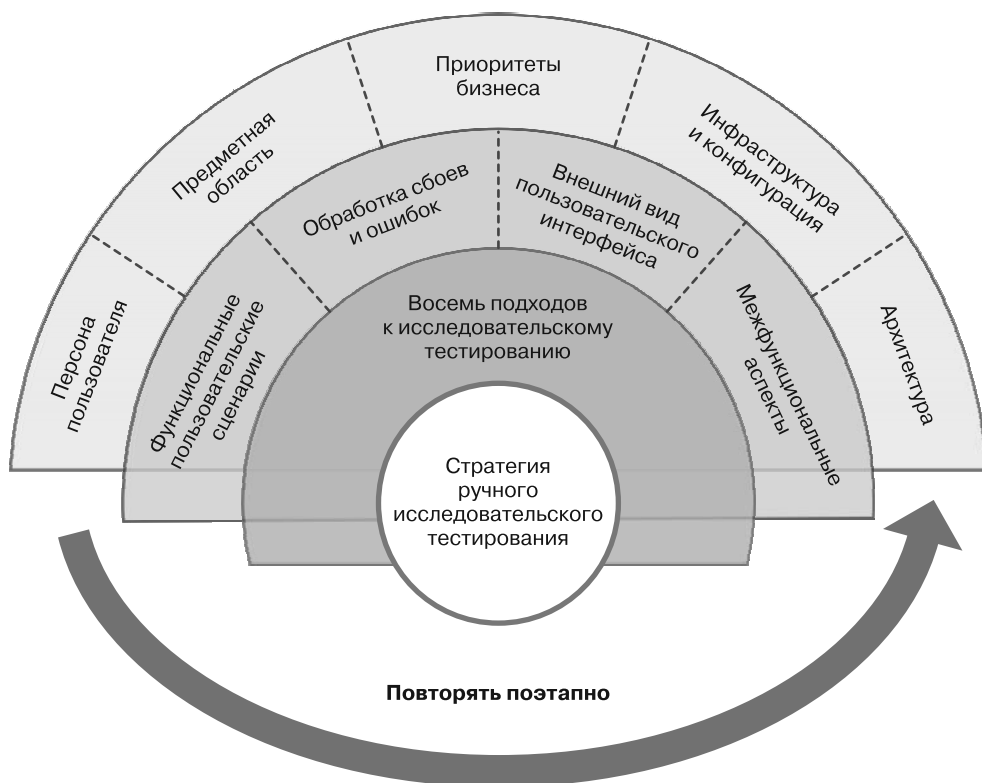


Рис. 2.7. Стратегия ручного исследовательского тестирования

Изучение приложения

Внешний полукруг соответствует изучению деталей приложения и делится на пять широких областей, на которых вы должны сосредоточиться. Сбор подробной информации об этих областях поможет приступить к исследовательскому тестированию, но, как упоминалось ранее, в ходе изучения вы обязательно найдете новую информацию о приложении.



Иногда исследовательское тестирование путают с бездумным тестированием, когда приложение тестируют со случайными входными данными, ничего не зная о функциональности. Важно отметить, что при исследовательском тестировании вы должны иметь точное представление о тестируемой функциональности и проводить его с прицелом на исследование неизвестного.

Вот краткое описание пяти основных областей, на которых нужно сосредоточиться, изучая приложение.

Персона пользователя

Персона (<https://oreil.ly/QtpCm>) — это вымышленный персонаж, представляющий группу конечных пользователей со схожими характеристиками. Персоны пользователей обычно определяют в начале проекта, чтобы иметь возможность учитывать их потребности на всех этапах жизненного цикла поставки, начиная с проектирования. Примером того, как персоны пользователей могут влиять на функциональность приложения, может служить сайт социальной сети. Его молодые пользователи могут рассчитывать на экстравагантный опыт, а пожилые — на ясное и понятное взаимодействие. Целью тестирования является изучение возможностей с точки зрения типичных представителей разных групп пользователей, поэтому жизненно важно знать, кто будет применять приложение, и исследовать особенности его восприятия и взаимодействия с ним каждого персонажа.

Предметная область

Каждая предметная область — социальные сети, транспорт, здравоохранение и т. д. — имеет свои особенности, порядок использования и терминологию или жаргон, которые необходимо изучить перед началом исследования. Электронная коммерция наглядно демонстрирует, насколько важно знать предметную область при тестировании. Например, после создания заказа преодолевает несколько этапов обработки: получение, оплата, подтверждение, отправка и т. д. В процессе его обработки происходит взаимодействие с многочисленными сторонами, такими как склад, где хранятся товары, транспортные компании, доставляющие их со склада покупателю, и производители, регулярно пополняющие

запасы на складе. Поэтому, наблюдая за поведением приложения во время исследовательского тестирования, необходимо знать, как исследовать все этапы обработки заказа в приложении. Вам будет сложно сделать это, не имея базовых знаний о предметной области.

Приоритеты бизнеса

Рассмотрим сценарий, в котором бизнес-приоритетом является разработка платформы (<https://oreil.ly/dEd9N>) для расширения и масштабируемости. В таких случаях простого тестирования функционального пользовательского сценария с применением только пользовательского интерфейса может быть недостаточно. Его необходимо изучить с точки зрения платформы, выяснить, насколько тесно связаны пользовательский интерфейс и веб-сервисы, можно ли отделить веб-сервисы, сделав их независимыми, чтобы получить возможность интеграции с другими системами, и разрешить многие другие подобные вопросы.

Инфраструктура и конфигурация

Как обсуждалось ранее, исследовательское тестирование предполагает вмешательство в тестовую среду для моделирования реальных сценариев, включая сбои. Наличие информации о том, какие компоненты приложения и где развернуты, а также о доступных настройках поможет обнаружить новые сценарии. Например, веб-сервисы могут предусматривать настройку ограничения максимального количества запросов в единицу времени (<https://oreil.ly/TYa3z>). В таком случае вам может потребоваться понаблюдать за поведением приложения при превышении этого предела. Сбор некоторой базовой информации об инфраструктуре и конфигурации, например об особенностях развертывания сервиса и базы данных (на одном компьютере или на нескольких машинах), о настройках ограничения частоты запросов, настройках шлюза API и т. п., поможет определить важные тестовые сценарии.

Архитектура приложения

Знание архитектуры приложения добавит ответвления к вашим сценариям в сеансе исследовательского тестирования. Например, если архитектура включает в себя веб-сервисы, то может потребоваться выполнить исследовательское тестирование не только пользовательского интерфейса, но и API (описано в разделе «Тестирование API» далее в этой главе). Аналогично, если приложение задействует потоки событий (обсуждаются в главе 5), важно изучить особенности асинхронных взаимодействий. Понимание архитектуры на высоком уровне поможет вам найти пути исследования внутренней интеграции компонентов, потоков данных между компонентами, интеграции с внешними сервисами и обработки ошибок. Некоторые из этих аспектов мы еще не раз будем обсуждать на протяжении всей книги.

Собрав достаточно информации об этих пяти областях, вы будете готовы приступить к реальному исследовательскому тестированию.

Даже если вас несколько ошеломило сказанное ранее, особенно та часть, где говорится об архитектуре и инфраструктуре, не беспокойтесь об этих деталях сейчас. Принято рассматривать исследовательское тестирование с функциональной точки зрения и постепенно учиться задавать все больше уточняющих вопросов.

Исследование по частям

Следующий полукруг на диаграмме (см. рис. 2.7), представляющей стратегию ручного исследовательского тестирования, показывает, что исследование должно производиться по частям.

В своей статье *Exploratory Testing Explained* (<https://oreil.ly/B7jaO>), опубликованной в 2003 году, Джеймс Бах определяет практику исследовательского тестирования как одновременное изучение, разработку тестов и их выполнение. Это одно из самых распространенных определений, существующих на сегодняшний день. Исследовательское тестирование — это выполнение ряда действий в приложении с одновременным наблюдением за его поведением и за счет этого получение более полного представления о приложении и постепенное его изучение. Этот процесс требует, чтобы мы все время были начеку и проводили *всесторонний анализ*. Будучи людьми, мы способны концентрироваться и по-настоящему вникать в суть чего-либо, только когда сосредотачиваемся на небольших объемах работы. Вот почему желательно изучать приложения по частям! Такими частями могут быть любые из ранее обсуждавшихся направлений или их ответвлений, такие как пользовательский сценарий, функция или межфункциональный аспект, например безопасность.

Однако отслеживать все эти направления и ответвления при углубленном изучении затруднительно. Одна из стратегий решения этой проблемы — использование ментальной карты¹, подобной той, что изображена на рис. 2.8. Ею можно поделиться со всей командой.

На этом этапе вам также могут понадобиться восемь подходов к исследовательскому тестированию, которые изображает внутренний полукруг на рис. 2.7.

¹ Ментальная карта — это средство визуализации, на котором фиксируются основные идеи вместе с их ответвлениями. Для составления ментальной карты можно использовать такие инструменты, как Coggle (<https://coggle.it>) и XMind (<https://www.xmind.net>).

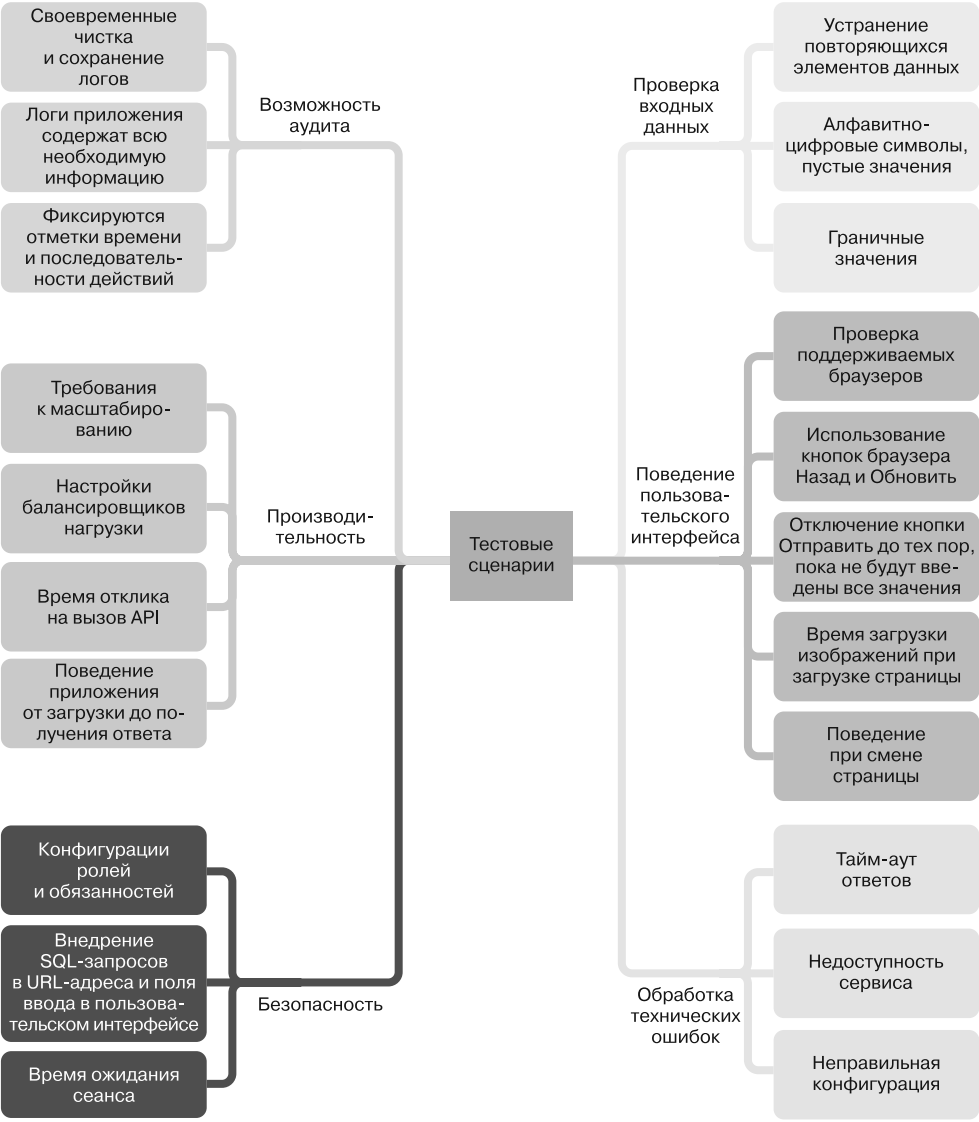


Рис. 2.8. Ментальная карта для исследовательского тестирования

Поэтапное повторение исследовательского тестирования

Исследовательское тестирование — не одноразовое мероприятие. Команда будет постоянно добавлять новый код и новые функции, меняющие поведение приложения, соответственно, новое поведение потребует повторного изучения. Рассмо-

трение исследовательского тестирования как непрерывного процесса позволяет структурировать область, которая должна быть изучена на определенном этапе. Например, некоторые Agile-команды практикуют тестирование в среде разработчика (*dev-box*), когда представители бизнеса и тестировщики вместе выполняют короткое ознакомительное тестирование создаваемой пользовательской истории на компьютере разработчика. В этом случае вы можете ограничиться тестированием только позитивных пользовательских сценариев, проверками данных и оценкой внешнего вида пользовательского интерфейса. Следующий этап, благоприятный для изучения, — тестирование пользовательской истории после реализации. В этом случае можно расширить область исследования, включив в нее проверку совместимости с разными типами браузеров и каких-то межфункциональных аспектов. Кроме того, некоторые Agile-команды регулярно *проверяют наличие ошибок*, когда все члены команды собираются вместе и изучают функции, разработанные к данному моменту. Наконец на этапе тестирования релиза можно сосредоточиться на межфункциональных аспектах, таких как производительность, надежность и масштабируемость, и изучить позитивные пользовательские сценарии и интеграцию компонентов на более высоком уровне. Заблаговременное планирование этапов исследовательского тестирования помогает командам получать постоянную обратную связь и, следовательно, пространство для постоянного совершенствования.



Исследовательское тестирование органично по своей природе. Поэтому вы можете обнаружить новые направления для исследования, которые не планировались ранее и которые могут занять время, отведенное на итерации. Этого следует ожидать. Советуем подумать, можно ли включить то или иное направление в очередной этап тестирования пользовательской истории или в поиск ошибок. Если можно, то просто запишите эту задумку, чтобы не забыть, и двигайтесь дальше.

Краткое резюме только что описанной стратегии: приступая к исследовательскому тестированию, сначала ознакомьтесь с деталями приложения, затем запишите возможные направления для изучения. А после этого продолжайте изучение различных направлений на всех этапах цикла поставки, чтобы обеспечить постоянную обратную связь с командой.

Упражнения

Мы довольно широко обсудили возможные подходы и стратегии. Чтобы применить их для исследования приложения, вам может потребоваться научиться работать с некоторыми инструментами, такими как SQL для изучения базы данных (обсуждается в главе 5), Postman для изучения API и т. д. В книге мы рассмотрим несколько таких инструментов, в частности, в этом разделе вы познакомитесь с инструментами исследовательского тестирования API и веб-интерфейса.

Тестирование API

Программный интерфейс приложения (Application Programming Interface, API) (<https://oreil.ly/jNiSY>) дает системам возможность взаимодействовать друг с другом. По сути, API абстрагируют тонкости системы и упрощают обмен информацией по сети в формате XML, JSON или в виде простого текста с помощью протокола HTTP. Для стандартизации обмена информацией были изобретены такие протоколы, как SOAP, и спецификации, такие как REST. В наши дни RESTful API более распространены, чем SOAP, и даже устаревшие системы, использующие SOAP, переписываются с учетом спецификаций REST. Чтобы вы могли лучше понять суть REST API, рассмотрим простое приложение электронной коммерции, подобное показанному на рис. 2.9, с тремя сервисами REST (Order — для управления заказами, Auth — для аутентификации и Customer — для обслуживания клиентов), пользовательским интерфейсом и базой данных.



Рис. 2.9. Пример приложения электронной коммерции с сервис-ориентированной архитектурой

Веб-сервис — это компонент, реализующий обособленную функциональность внутри приложения. Так, сервис Order в нашем примере может отвечать за управление заказами (создание, обновление и удаление), а сервис Customer — за хранение и предоставление сведений о клиентах. Такая организация упрощает обмен информацией, потому что другие компоненты приложения, например пользовательский

интерфейс или иные сервисы, могут обратиться к API соответствующего сервиса и получить необходимую им информацию.



Сервис-ориентированная архитектура — это архитектура, в которой основные функции приложения реализованы в виде веб-сервисов (см. рис. 2.9).

В качестве примера предположим, что конечный пользователь оплачивает заказ через пользовательский интерфейс приложения. Как показано в примере 2.1, этот интерфейс немедленно отправит сервису Order запрос на создание заказа со всеми необходимыми деталями, а тот обработает запрос и вернет ответ пользовательскому интерфейсу. Последний в этом контексте называется *клиентом*.

Пример 2.1. Пример REST-запроса и REST-ответа

// Запрос

```
POST method: http://eCommerce.com/orders/new
{
  "name": "V-Neck Tshirt",
  "sku": "ABCD1234",
  "color": "Red",
  "size": "M"
}
```

// Ответ

```
Status Code: 200 OK
Response Body:
{
  "Msg": "successfully created",
  "ID": "Order1234227891"
}
```

Рассмотрев запрос из примера 2.1, можно заметить, что он обращается к API `/orders/new` с помощью HTTP-метода POST. Обычно метод POST используется для создания или добавления новой информации, а метод GET — для получения информации, например списка заказов, сделанных покупателем. Также существуют методы PUT и DELETE, которые применяются для выполнения операций обновления и удаления соответственно. В тело запроса упаковываются сведения о заказе — в данном случае название товара, единица складского учета (Stock-Keeping Unit, SKU), цвет и размер — в виде объекта JSON. Вся эта структура называется *контрактом*. Если клиент не соблюдает установленный контракт, то сервис не будет обрабатывать запрос.

Ответ также оформляется в соответствии с контрактом: он включает код состояния, сообщающий об успехе или неудаче операции, и тоже может иметь тело

с дополнительной информацией об операции. В примере 2.1 код состояния ответа — `200 OK` — сообщает об успехе, а в теле ответа содержится сообщение `successfully created` (успешно создано) вместе с идентификатором заказа, сгенерированным сервисом `Order`. Получив этот ответ, пользовательский интерфейс перенаправит пользователя на страницу подтверждения и отобразит на ней идентификатор заказа. Обратите внимание на то, что все эти действия будут происходить синхронно, то есть пользовательский интерфейс будет ждать, пока не получит ответ, и только потом перейдет на страницу подтверждения заказа.

Теперь, получив представление о работе API, вы можете спросить: зачем нам изучать API, если можно протестировать функциональность создания заказов из веб-интерфейса? Ответ прост: в настоящее время API сами стали продуктами! Если отвечать более развернуто, то дело в том, что API охватывают всю бизнес-логику и проверки, что делает их автономными продуктами, которые могут использоваться другими внутренними и внешними компонентами. Например, в рамках развития нашего гипотетического бизнеса электронной коммерции мы могли бы создать новое мобильное приложение для покупок или портал поддержки клиентов и повторно применить те же API создания заказов и обслуживания клиентов. Могли бы даже запустить совершенно новый бизнес и повторно задействовать API сервиса аутентификации для реализации функций входа в новом продукте. Поэтому изучение API как отдельных продуктов очень важно в современном цифровом мире.

Вот некоторые направления исследований, на которые следует обратить внимание при изучении API, помимо основной бизнес-логики.

Валидация контракта запроса

Проверка должна выполняться так, чтобы, получив новый заказ с недопустимыми форматами данных, например, от клиента-мошенника, сервис `Order` отклонил запрос.

Аутентификация

Из соображений безопасности в большинстве случаев API защищаются с помощью некоторых механизмов аутентификации, таких как отправка токена (длинной зашифрованной строки) в заголовке запроса. Это важное направление для исследования.

Разрешения

API могут иметь ограничения на операции, выполняемые для клиентов. Например, администратору может быть разрешено редактировать существующий заказ, а менеджеру по работе с клиентами — только просматривать его.

Обратная совместимость

Иногда по мере развития продукта может потребоваться изменить контракты API. Но, поскольку API часто используются уже существующими клиентами,

может потребоваться создавать и поддерживать новые версии параллельно со старыми. Приложение необходимо тестировать с обеими версиями API.

Коды состояния HTTP

Коды состояния, возвращаемые в случае сбоев, должны соответствовать причине сбоя. В табл. 2.4 перечислены наиболее распространенные коды состояния.

Таблица 2.4. Коды состояния HTTP и их значение

Код состояния	Значение
200 OK	Сообщает об успешной обработке запроса GET, PUT или POST
201 Created (Создано)	Сообщает, что успешно создан новый объект, например новый заказ
400 Bad Request (Неверный запрос)	Сообщает, что запрос неправильно сформирован
401 Unauthorized (Не авторизован)	Сообщает, что клиенту запрещен доступ к запрошенному ресурсу и ему следует повторить запрос, включив необходимые учетные данные
403 Forbidden (Запрещено)	Сообщает, что запрос составлен верно и клиент аутентифицирован, но ему по какой-то причине запрещен доступ к запрошенным странице или ресурсу
404 Not Found (Не найдено)	Сообщает, что запрошенный ресурс сейчас недоступен
500 Internal Server Error (Внутренняя ошибка сервера)	Сообщает, что запрос составлен верно, но сервер не может его обработать, возможно, из-за внутренней ошибки
503 Service Unavailable (Сервис недоступен)	Сообщает, что сервер не работает (например, находится на техническом обслуживании)

Для изучения всех этих направлений вам потребуются инструменты, и в следующих разделах я представлю некоторые из них.

Postman

Postman (<https://www.postman.com>) — широко известный инструмент для тестирования API. Файлы для установки десктопной версии доступны бесплатно, также можно попробовать веб-версию. Здесь я кратко расскажу о десктопном приложении.

1. Загрузите установочный файл для вашей ОС с официального сайта (<https://www.postman.com/downloads>).
2. Запустите Postman и выберите **New ▶ HTTP Request** (Создать ▶ HTTP-запрос). Откроется диалоговое окно создания нового запроса.

- 3. В браузере выполните поиск в Google по запросу «исследовательское тестирование», затем скопируйте URL и вставьте его в поле URL в диалоговом окне создания запроса в Postman (рис. 2.10). Обратите внимание на то, что в раскрывающемся списке рядом с этим полем автоматически выбирается HTTP-метод GET.
- 4. На вкладке Params (Параметры) можно увидеть автоматически заполненные параметры поискового запроса Google. Параметр запроса q будет содержать строку исследовательское+тестирование. Вы можете изменить ее, указав любое другое ключевое слово для поиска.
- 5. Нажмите кнопку Send (Отправить), чтобы выполнить запрос.
- 6. В ответ получите код состояния ответа, заголовки, тело и cookie-файлы, которые появятся на нижней панели.

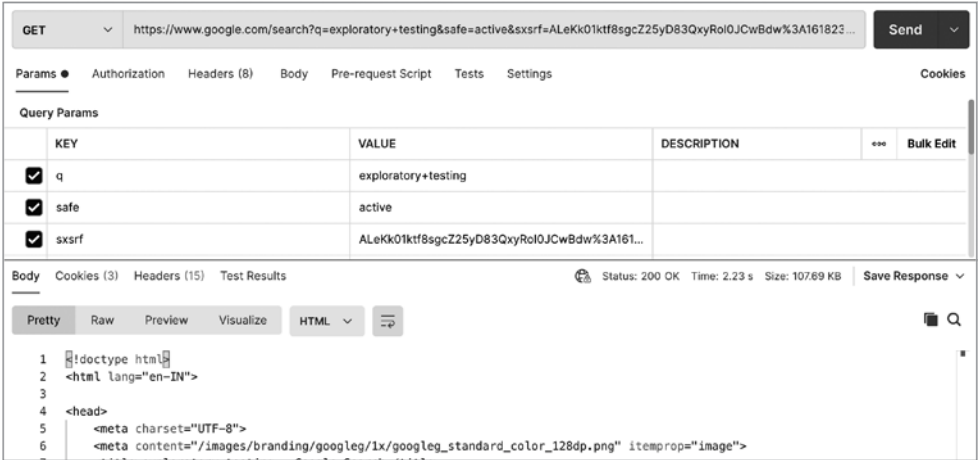


Рис. 2.10. Создание нового запроса и проверка ответа в Postman

В данном случае ответом является страница HTML. Если нажать кнопку Preview (Предварительный просмотр), то в браузере откроется та же самая страница с результатами поиска. Однако во многих случаях в ответ возвращается объект JSON, как в примере 2.1: пользовательский интерфейс Postman проанализирует JSON и отобразит нужную информацию.

Поиск в Google осуществляется с помощью запросов GET, однако запросы POST создаются аналогично: выберите метод POST в раскрывающемся списке, введите запрос к API в поле URL и тело запроса на вкладке Body (Тело) и нажмите кнопку Send (Отправить), чтобы получить ответ. Если хотите еще немного попрактиковаться, то в Any API (<https://any-api.com>) найдете сводный список с 1400 общедоступными REST API, которые можно опробовать.



По умолчанию Postman сохраняет историю выполняемых вами действий в облаке под вашей учетной записью Postman. Благодаря этому можно начать работать на одном компьютере и продолжить на другом (<https://oreil.ly/Yl5v9>). Однако убедитесь в том, что синхронизация с облаком Postman не нарушает никаких соглашений с клиентами о неразглашении или вашей внутренней политики.

Postman предоставляет еще несколько возможностей для изучения API. Далее перечислены некоторые из наиболее востребованных.

- Для целей аутентификации вместе с запросом можно отправить токен, добавив его на вкладке **Authorization** (Авторизация). Для примера попробуйте добавить туда недопустимые строки, чтобы убедиться, что запрос не выполняется.
- Также вместе с запросом можно отправить cookie-файлы, добавив их на вкладке **Cookies**, которая расположена под кнопкой **Send** (Отправить).
- Postman фиксирует время, потребовавшееся для получения ответа. Оно выводится рядом с кодом состояния (рис. 2.10). Это поможет быстро выявить снижение производительности для разных входных данных.
- Чтобы не создавать запросы вручную, можно импортировать спецификации API из Swagger, OpenAPI и так далее вместе с их ссылками.
- В дополнение к REST-сервисам Postman поддерживает тестирование сервисов GraphQL и SOAP.

WireMock

WireMock (<http://wiremock.org>) — это инструмент для создания и изменения *заглушек* — программных компонентов, имитирующих поведение других компонентов. Заглушки особенно полезны при разработке и тестировании сложных распределенных приложений, в которых еще не все сервисы готовы. Команды согласовывают контракт, поддерживаемый сервисами, и могут продолжить разработку, создавая заглушки сервисов. Заглушки получают путем их явного программирования для обработки определенных запросов и возврата определенного ответа. Эту функцию можно использовать при исследовательском тестировании для настройки различных позитивных и негативных тестовых сценариев. Конечно, как только все компоненты будут готовы, важно еще раз протестировать сквозную функциональность с реальными компонентами.



Настройку сервера-заглушки и приложения для использования заглушки могут выполнить DevOps-инженер или разработчики из команды. Однако тестировщикам необходимо знать, как изменять заглушки, чтобы моделировать тестируемые ситуации. Специально для этой цели я включила упражнение.

Чтобы показать, как использовать WireMock, вернемся к примеру приложения электронной коммерции. Предположим, что у нас пока нет фактического платежного сервиса, но мы знаем контракт запроса и ответа конечной точки `/makePayment`,

которую пользовательский интерфейс применяет для отправки платежей. Чтобы изучить различные тестовые сценарии взаимодействий, нужно настроить заглушку конечной точки `/makePayment` с позитивными и негативными ответами. Соответствующие шаги описаны далее.

1. Загрузите JAR-файл WireMock с официального сайта (<https://oreil.ly/qsBOh>).
2. Откройте терминал и выполните следующую команду, чтобы запустить загруженный вами файл JAR:

```
$ java -jar wiremock-jre8-standalone-xxx.jar
```

Эта команда запустит сервер WireMock на порте 8080.

3. Чтобы получить новую заглушку, создайте контракт API `/makePayment`, как показано в примере 2.2, и отправьте запрос POST конечной точке `http://localhost:8080/__admin/maps/new` с помощью Postman. То есть в диалоговом окне создания нового запроса в Postman выберите в раскрывающемся списке HTTP-метод POST, введите URL в поле URL и JSON-код из примера 2.2 на вкладке Body ▶ raw (Тело ▶ неформатированное) и нажмите кнопку Send (Отправить).

Пример 2.2. Пример заглушки WireMock

```
{
  "request": {
    "method": "POST",
    "url": "/makePayment"
  },
  "response": {
    "status": 200,
    "body": "Payment Successful"
  }
}
```

4. Теперь убедитесь, что заглушка работает, создав еще один запрос POST для перехода по URL `http://localhost:8080/makePayment`. Вы должны получить ответ с кодом состояния 200 OK и сообщением `Payment Successful`, как описано в заглушке. Получив этот ответ, наш воображаемый пользовательский интерфейс должен отобразить страницу подтверждения заказа.
5. Теперь измените заглушку так, чтобы она возвращала ответ с сообщением об ошибке. Для этого измените тело ответа в примере 2.2, как показано далее, и отправьте запрос POST той же конечной точке `/mappings/new`:

```
"response": {
  "status": 401,
  "body": "Payment Unauthorized"
}
```

Получив такой ответ, пользовательский интерфейс должен вывести сообщение об ошибке.

Аналогично можно настроить и другие тестовые сценарии (недопустимые запросы, недоступность сервиса и т. д.) с соответствующими кодами состояния в теле ответа и понаблюдать, как их обрабатывает пользовательский интерфейс. Как видите, заглушки помогают при исследовательском тестировании API, когда реальные сервисы недоступны для тестирования.

Теперь перейдем к инструментам исследовательского тестирования веб-интерфейса.

Тестирование веб-интерфейса

В этом разделе освещаются три основных инструмента тестирования веб-интерфейса: браузеры, Bug Magnet и Chrome DevTools.

Браузеры

Первым и главным инструментом для изучения веб-интерфейса является браузер. Во время тестирования желательно охватить не менее 85 % пользовательской базы приложения. На момент написания этих строк самые последние статистические данные о популярности браузеров, приведенные на gs.statcounter.com (<https://gs.statcounter.com>), показывают, что доля Chrome составляет около 64,5 %, за ним следуют Safari — 18,8 %, Edge — 4,05 %, Firefox — 3,4 % и Samsung Internet — 2,8 %. Учитывая эти данные, желательно включить в тестирование Chrome и Safari. Третье место часто попеременно занимают Edge и Firefox, поэтому рекомендуется включить оба. Все эти браузеры доступны для любых ОС.



Иногда может потребоваться протестировать старые браузеры, такие как Internet Explorer 11 или Edge Legacy, даже притом что Microsoft официально прекратила поддержку этих версий. Один из способов сделать это — загрузить виртуальную машину Windows (<https://oreil.ly/IOUWS>) на свой компьютер.

Также можно использовать облачные платформы тестирования, такие как BrowserStack (<https://www.browserstack.com>) и Sauce Labs (<https://saucelabs.com>), которые избавят вас от необходимости устанавливать разные версии браузеров и ОС на локальные компьютеры. Они предоставляют виртуальный доступ к браузерам на разных ОС за определенную плату. Процесс прост: оплатите подписку (имеются и бесплатные пробные версии), войдите на портал, выберите комбинацию версии браузера и ОС (рис. 2.11) и протестируйте свое приложение.

BrowserStack поддерживает также локальное тестирование (<https://oreil.ly/6DLat>) частных приложений, размещенных в средах отделов контроля качества или

средах обкатки. В зависимости от потребностей может оказаться целесообразным подписаться на такую услугу, благодаря которой вы сможете протестировать свой веб-интерфейс с широким спектром старых браузеров.

Quick Launch							
Android	89 Latest	11 Latest	87 Latest	89 Latest	75 Latest	14.12 Latest	5.1 Latest
iOS	90 Beta	10	88 Beta	90 Beta	76 Dev		5
Windows	91 Dev	9	86	91 Dev	74		4
10	88	8	85	88	73		
8.1	87		84	87	72		
8	86		83	86	71		
7	85		82	85	70		
XP	84		81	84	69		
Mac	83		80	83	68		
	81		79	81	67		
	1 more		77 more	66 more	58 more		

Рис. 2.11. BrowserStack и другие подобные сервисы позволяют выполнять тестирование с использованием разных комбинаций браузеров и ОС

Bug Magnet

Bug Magnet (<https://bugmagnet.org>) — это плагин браузера для Chrome и Firefox, который позволяет тестировать крайние случаи. Он предоставляет список типичных тестовых сценариев и соответствующие значения, которые необходимо ввести в поля ввода в приложении для каждого тестового сценария. Этот инструмент действует подобно чек-листу для исследовательского тестирования. Чтобы опробовать его, выполните следующие действия.

- 1. Установите плагин (<https://oreil.ly/5sbqz>) в браузер Chrome.
- 2. Откройте страницу поиска Google (<https://www.google.com>) и щелкните правой кнопкой мыши на текстовом поле ввода строки поиска.
- 3. В открывшемся контекстном меню вы увидите пункт Bug Magnet (рис. 2.12). Как видите, здесь предлагается проверить множество крайних случаев, из которых можно выбрать один. Например, выберите Names ▶ Name Length (Имена ▶ Длина имени) и первое имя. Длинное имя появится в текстовом поле ввода поиска Google. Если в вашем приложении проверяется длина входной строки, то должно появиться сообщение об ошибке.



В Bug Magnet есть еще несколько эвристик для исследовательского тестирования (<https://oreil.ly/O29Em>), которые помогут не пропустить важные тестовые сценарии. Они особенно полезны для новичков.



Рис. 2.12. Плагин Bug Magnet можно использовать как руководство в ходе ручного исследовательского тестирования

Chrome DevTools

Chrome DevTools (<https://oreil.ly/T0rlU>) — универсальный набор, подобный швейцарскому армейскому ножу. Он включает в себя множество инструментов, полезных в исследовательском тестировании, тестировании безопасности, производительности и многих других аспектов. Этот инструмент часто будет встречаться вам на протяжении всей книги. Чтобы получить представление о его возможностях, выполните следующие действия.

1. Откройте браузер Chrome и выполните поиск по строке «исследовательское тестирование».
2. Щелкните правой кнопкой мыши на странице с результатами поиска и выберите пункт **Inspect** (Просмотреть код). После этого сразу откроется панель с инструментами разработчика. Открыть панель DevTools можно также с помощью комбинации клавиш **Cmd+Option+C** или **Cmd+Option+I** в macOS либо **Shift+Ctrl+J** в Windows.

Здесь можно изучить множество разных аспектов, в том числе перечисленные далее.

Ошибки на странице

Как показано на рис. 2.13, на вкладке **Console** (Консоль) отображаются ошибки, обнаруженные на веб-странице. Обычно она не должна содержать ошибок, поэтому рекомендуется проверять эту вкладку после перехода на каждую новую страницу тестируемого приложения. Ошибки, перечисленные здесь, могут

помочь устранить любые проблемы, обнаруженные на веб-странице. Например, если на ней отсутствует какое-то изображение, откройте вкладку **Console** (Консоль) и включите обнаруженное сообщение об ошибке в баг-репорт.

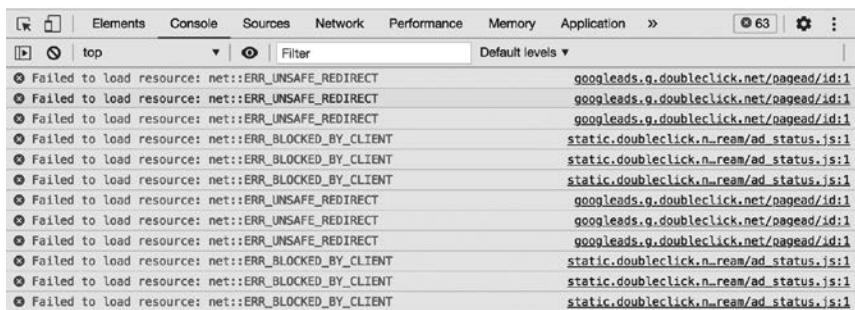


Рис. 2.13. Вкладка **Console** (Консоль) со списком ошибок, обнаруженных на странице

Количество запросов, отправленных со страницы

Иногда из-за ошибок в логике приложения веб-страница может посылать в API много нежелательных вызовов (<https://oreil.ly/wUswC>) и работать медленно. Обнаружить такие проблемы можно на вкладке **Network** (Сеть), где слева внизу показано общее количество запросов, отправленных с этой страницы.

Поведение при первом использовании

При повторном тестировании одного и того же приложения некоторые ресурсы (например, изображения на веб-страницах) кэшируются. Поэтому, если изображение изменится во время разработки, то эти изменения могут не отображаться. Установите флажок **Disable cache** (Отключить кэш) на вкладке **Network** (Сеть), чтобы очистить кэш, и обновите страницу. Имейте в виду, что кэш работает точно так же и у конечных пользователей, поэтому данный флажок поможет вам изучить опыт пользователя при первом контакте с приложением.

Поведение пользовательского интерфейса в медленных сетях

Чтобы изучить опыт конечного пользователя в процессе работы в сети с ограниченной пропускной способностью, можно выбрать скорость сети на вкладке **Network** (Сеть) и понаблюдать за поведением пользовательского интерфейса. Как показано на рис. 2.14, рядом с флажком **Disable cache** (Отключить кэш) находится раскрывающийся список, позволяющий моделировать условия работы в сети 2G, 3G и 4G. Выберите интересующий вас вариант, очистите кэш браузера и перезагрузите страницу. DevTools отобразит серию скриншотов, иллюстрирующих процесс загрузки приложения в сети с указанной пропускной способностью. Прогрессивные веб-приложения (обсуждаются в главе 11) работают даже в автономном режиме, и раскрывающийся список настройки

пропускной способности сети тоже содержит вариант выбора автономного режима для проверки такого поведения.

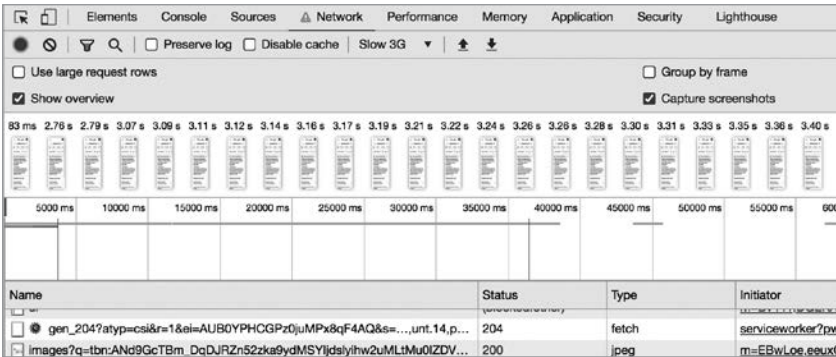


Рис. 2.14. Управление пропускной способностью сети в Chrome DevTools

Интеграция пользовательского интерфейса и API

На вкладке Network (Сеть) фиксируются все сетевые вызовы, выполненные веб-страницей, в том числе вызовы веб-сервисов из пользовательского интерфейса, включая заголовки запросов и ответов (и токены аутентификации), параметры запросов, ответы и другую полезную информацию о каждом сделанном запросе (рис. 2.15).

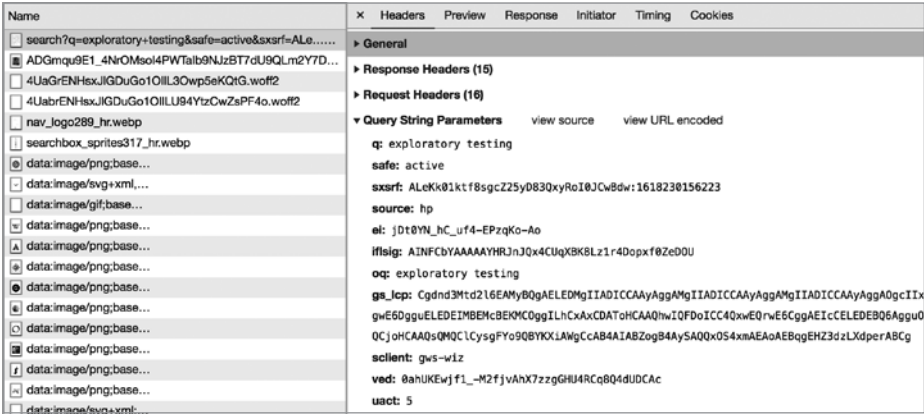


Рис. 2.15. Информация о запросах и ответах на вкладке Network (Сеть)

С помощью сведений о запросе/ответе можно изучать интеграцию пользовательского интерфейса и API. Например, можно проверить, передает ли этот интерфейс правильные параметры запроса, введенные конечным пользователем.

Также можно наблюдать за поведением пользовательского интерфейса при получении различных ответов от сервиса. Например, узнать, отображает ли он сообщение об ошибке «Элемент недоступен», получая от API код состояния 404.

Поведение при отключении сервиса

Для моделирования случаев сбоев запросов можно заблокировать конкретный запрос на вкладке **Network** (Сеть) и понаблюдать за поведением пользовательского интерфейса. Например, найдите на вкладке **Network** (Сеть) URL, соответствующий первому изображению на странице с результатами поиска в Google по строке «исследовательское тестирование». Щелкните на нем правой кнопкой мыши, выберите в меню **Block Request URL** (Заблокировать URL запроса) и перезагрузите страницу. Соответствующее изображение не загрузится в пользовательский интерфейс. Эту функцию можно применять для тестирования сценария «отключение сервиса» без фактического его отключения.

Cookie-файлы

Cookie-файлы в основном используются для хранения информации о сеансе. На вкладке **Application** (Приложение) отображаются список сохраненных cookie-файлов и их содержимое (рис. 2.16). Здесь можно редактировать или удалять cookie-файлы в процессе тестирования и наблюдать за поведением приложения.

Подробнее узнать обо всех функциях Chrome DevTools можно на официальном сайте (<https://oreil.ly/J34ry>).

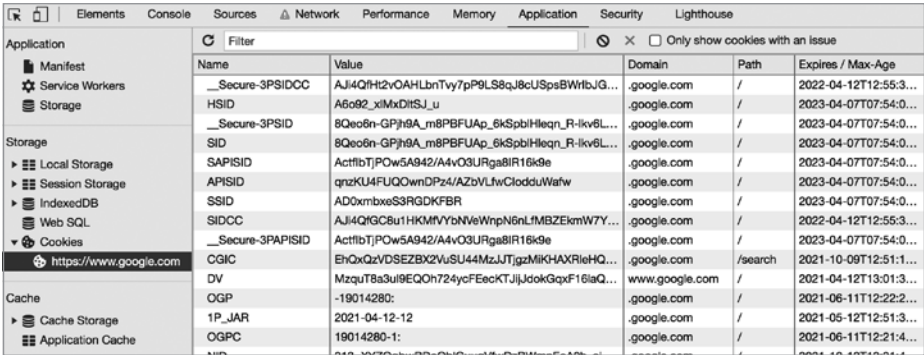


Рис. 2.16. Cookie-файлы можно редактировать и удалять на вкладке Application (Приложение)

Теперь у вас есть несколько инструментов, с помощью которых можно изучать API и веб-интерфейсы. Но на пути к достижению целей исследовательского тестирования остается еще одна ключевая тема — поддержание хорошей гигиены в тестовой среде. Обсудим этот вопрос далее.

Перспективы: гигиена тестовой среды

Тестовая среда — это настоящая игровая площадка, где тестировщики применяют свои навыки исследовательского тестирования, и недостаточная поддержка этой среды напрямую влияет на тестировщиков и результаты их деятельности. Далее описаны некоторые признаки плохой технической поддержки, с которыми вы можете столкнуться, их влияние и способы устранения недостатков.

Общие и выделенные тестовые среды

В больших командах одна тестовая среда часто используется несколькими группами, что серьезно ограничивает для тестировщиков возможность вмешиваться в настройки среды, если это необходимо для исследования разных направлений. Например, чтобы на короткое время отключить какой-то сервис, им необходимо получить согласие других групп. Хуже того, нужно координировать с ними свои действия или ждать следующего запланированного развертывания, которое может проводиться раз в день или раз в неделю, чтобы изучить поведение последней версии кода. Наличие выделенной (индивидуальной) среды тестирования, по крайней мере с компонентами, входящими в компетенцию отдельной группы, даст тестировщикам свободу для более смелых исследований.

Гигиена развертывания

Если у команды есть собственная выделенная среда, то лучше всего запускать новые развертывания вручную вместо использования автоматизированного конвейера непрерывной интеграции, потому что это позволит тестировщикам свободно менять конфигурацию среды. Кроме того, сборка должна быть доступна для развертывания в конвейере CI только после завершения этапа автоматизированного тестирования. Это делается для того, чтобы не допустить появления в последней версии кода дефектов, способных помешать исследовательскому тестированию. Подробнее о CI и стратегиях развертывания мы поговорим в главе 4.

Кроме того, тестовая среда должна быть настроена так, чтобы максимально походить на промышленную среду с брандмауэрами, разграниченными уровнями/компонентами, настройками ограничения скорости и т. д. Только тогда можно тщательно изучить причины неудач тестов, обсуждавшихся ранее.

Гигиена тестовых данных

Тестовые данные входят в компетенцию тестировщиков, и они должны следовать определенным правилам, чтобы гарантировать, что не возникнет непреднамеренных препятствий в исследованиях. В частности, будьте осторожны с устаревшими данными и конфигурациями, начиная тестировать новую функцию в том же развертывании. Чтобы избежать подобных осложнений, сделайте обязательным развертывание новой сборки в начале тестирования

каждой новой пользовательской истории (при условии, что новое развертывание удалит старые данные и конфигурации и приведет приложение в состояние, соответствующее новой функции). Другой вариант — создать для каждой пользовательской истории новый набор тестовых данных, например новую учетную запись пользователя, вместо того чтобы проводить исследования с существующими учетными записями, которые могут находиться в разных состояниях.

Создание тестовых данных может оказаться сложной задачей при наличии сотен связанных таблиц. В таких случаях можно в новом развертывании удалить старые данные и заменить их стандартным набором тестовых данных или написать SQL-скрипт, создающий соответствующие новые тестовые данные в процессе развертывания. Другая возможность — анонимизация промышленных данных при их использовании в тестовой среде, однако в этом случае необходимо принять особые меры предосторожности, чтобы не допустить просачивания туда конфиденциальных данных.

Автономные команды

Часто доступ к тестовой среде ограничен. Члены группы могут не иметь учетных данных для входа или необходимых разрешений для обновления конфигураций, просмотра логов приложений или настройки заглушек — для выполнения подобных действий им необходимо обратиться за помощью к команде DevOps или занимающейся обслуживанием системы. Это сильно мешает исследовательскому тестированию, когда тестировщикам требуется возможность доступа ко всем компонентам приложения. Обеспечение автономности команды и доступности всего, что ей нужно, поможет сократить задержки из-за внешних зависимостей и обеспечить бесперебойную доставку.

Настройка сторонних сервисов

Обычно сторонние сервисы не включаются в настройку тестовой среды, при этом предполагается, что интеграцию можно протестировать непосредственно в рабочей среде. Это может привести к нежелательным блокировкам, особенно если проблемы обнаруживаются на поздних стадиях цикла доставки. Следовательно, при настройке тестовой среды важно гарантировать наличие какого-то способа изучения интеграции со сторонними сервисами, либо используя заглушки, либо платя за ограниченный доступ к этим сервисам.

Теперь, обсудив, как и зачем проводить ручное исследовательское тестирование, важно подчеркнуть, что оно остается искусством, зависящим от аналитических навыков человека и его наблюдательности. Из-за этого индивидуалистского характера исследовательского тестирования не существует конкретного способа проверки его результатов. Другими словами, сегодня ваш аналитический мозг может открыть новые направления, которые приведут к обнаружению ошибок, а завтра — не сделать этого. Из-за такой непредсказуемости важно соблюдать дисциплину исследовательского тестирования, придерживаясь концепций, упомянутых в этой главе.

Ключевые выводы

- Ручное исследовательское тестирование предполагает изучение тестируемого приложения, чтобы понять его поведение, что в конечном итоге может привести к обнаружению новых сценариев пользователей и ошибок в них.
- Ручное исследовательское тестирование отличается от ручного тестирования тем, что последнее предполагает проверку списка спецификаций, тогда как первое опирается на индивидуальные аналитические навыки и наблюдательность.
- Исследовательское тестирование объединяет потребности бизнеса, техническую реализацию и точку зрения конечного пользователя, одновременно подвергая сомнению то, что известно как истинное со всех этих точек зрения.
- Мы обсудили восемь подходов к исследовательскому тестированию, которые могут помочь в структурировании мыслительных процессов тестировщика и создании осмысленных тестовых сценариев.
- Стратегия ручного исследовательского тестирования предполагает изучение деталей поведения приложения в пяти широких областях и проведение исследований в четырех основных направлениях, таких как: функциональные пользовательские сценарии, обработка сбоев и ошибок, внешний вид пользовательского интерфейса и межфункциональные аспекты.
- Исследовательское тестирование должно проводиться непрерывно. Мы можем запланировать его повторение на разных этапах жизненного цикла поставки, например при тестировании на сервере разработки, анализе пользовательских историй, выявлении ошибок и тестировании выпускаемой версии.
- Для изучения различных аспектов поведения приложения вам может потребоваться научиться применять новые инструменты. В этой главе обсуждались инструменты исследовательского тестирования API и веб-интерфейса: Postman, WireMock, Bug Magnet и Chrome DevTools.
- Тестовая среда — это площадка для ручного исследовательского тестирования, и ее поддержка имеет решающее значение для достижения целей исследовательского тестирования. Мы обсудили некоторые распространенные проблемы, возникающие при обслуживании тестовой среды, и способы их решения.
- Ручное исследовательское тестирование — это очень индивидуальный процесс, основанный на аналитических навыках и наблюдательности тестировщиков. Структурирование подхода к исследовательскому тестированию жизненно важно для получения результатов.

Автоматизированное функциональное тестирование

Включите свой автопилот!

Автоматизированное тестирование — это практика применения инструментов вместо людей для выполнения в приложении действий, которые обычно выполняет пользователь, и проверки ожидаемого поведения. Эта практика существует с 1970-х годов, а методы и инструменты в этой области постоянно развиваются вместе с программным обеспечением. Приведу несколько примеров. В 1970-х годах программные приложения преимущественно писались на FORTRAN, а для автоматизированного тестирования использовался инструмент RXVP. В 1980-х, когда началась эпоха развития персональных компьютеров (ПК), для автоматизированного тестирования был разработан инструмент AutoTester. В 1990-х, когда начался расцвет Всемирной паутины, появились такие инструменты автоматизации тестирования, как Mercury Interactive и QuickTest, также был изобретен инструмент автоматизированного нагрузочного тестирования Apache JMeter. Благодаря развитию Интернета в 2000-х годах появился Selenium, и с тех пор количество инструментов автоматизированного тестирования росло непрерывно. В настоящее время у нас есть даже инструменты автоматизированного тестирования на базе искусственного интеллекта и машинного обучения, которые обогащают общий опыт автоматизации тестирования.

Такое нововведение было вызвано несколькими ключевыми наблюдениями: автоматизированное тестирование значительно снижает стоимость этой процедуры и позволяет разработчикам ПО быстрее получать информацию о качестве приложений, чем при ручном тестировании. Чтобы убедиться в верности этого утверждения, рассмотрим сценарий, в котором на протяжении всего цикла разработки выполняется только ручное тестирование, а затем сценарий с использованием автоматизированного тестирования в той же ситуации. Допустим, в среднем для каждой функции приложения создано 20 тестовых сценариев и на проверку каждого из них вручную уходит около двух минут, то есть на тестирование одной функции требуется 40 минут. Всякий раз, когда разрабатывается новая функция, вам необходимо протестировать ее интеграцию с существующими функциями, а также убедиться, что работа существующих функций не нарушается новыми

изменениями, — эта практика называется *регрессионным тестированием*. Отказ от выполнения регрессионного тестирования на достаточно раннем этапе чреват обнаружением ошибок интеграции только во время тестирования релиза, то есть на слишком поздней стадии цикла, что может привести к нарушению сроков выпуска новой версии. Итак, в нашем примере регрессионное тестирование вместе с тестированием новых функций займет 80 минут при наличии второй функции, 120 минут — третьей и т. д.

Довольно быстро, когда в приложении появятся 15 функций, вам придется запланировать на тестирование 600 минут. Хуже того, иногда зрелому приложению приходится работать с разными версиями сервисов. Время тестирования будет увеличиваться пропорционально количеству версий сервисов, которые нужно поддерживать. Например, при необходимости поддержки двух версий сервиса время тестирования приложения возрастет до 1200 минут. Кроме того, если обнаружатся ошибки, то в зависимости от их характера (например, для устранения ошибки может потребоваться изменить схему БД) вы можете потратить еще 1200 минут на тестирование приложения перед запуском в эксплуатацию! В каждом следующем цикле время тестирования будет увеличиваться.

Компании, экономящие на затратах на автоматизированное тестирование, борются с этой проблемой, расширяя возможности ручного тестирования, но и в этом случае они получают обратную связь с большей задержкой, чем при автоматическом тестировании. Например, для нашего гипотетического приложения, даже если его одновременно тестируют 12 человек, на полноценную проверку всех аспектов потребуются в сумме 100 минут, тогда как автоматизированные тесты, запускаемые на соответствующих уровнях, могут выполняться намного быстрее и давать обратную связь значительно раньше. Важно не забывать и то, что при наличии автоматизированных тестов вам не нужно собирать 12 товарищей по команде в полную, чтобы протестировать срочное исправление производственного дефекта перед выпуском. И даже если вы пойдете на такой шаг, то ручное тестирование не дает полных гарантий от человеческих ошибок, поскольку оно во многом зависит от качества документации и выполнения тестовых сценариев.

Конечно, создание автоматизированных тестов и их регулярное выполнение сопряжены с определенными затратами. Однако их необходимо сопоставлять с выгодами, которые дает быстрая и частая доставка продукта на рынок, затратами на ручное тестирование (с точки зрения времени и возможностей) и уверенностью, которую оно дает команде во время разработки и решения проблем.

Таким образом, для создания высококачественного продукта можно порекомендовать использовать как ручное, так и автоматизированное тестирование, а также выработать разумную стратегию их балансирования: выбор только какого-то одного вида тестирования — не лучший вариант. Проще говоря, стратегия может быть такой: применять ручное *исследовательское* тестирование для выявления новых тестовых сценариев и далее автоматизировать их проверку для регрессионного тестирования.

Ручное исследовательское тестирование мы обсудили в главе 2, а цель этой главы — познакомить вас с приемами эффективного автоматизированного функционального тестирования веб-приложений на всех уровнях приложения. Далее я представлю стратегию автоматизированного функционального тестирования, которая поможет вашей команде быстрее получать обратную связь, и покажу, как настраивать и использовать инструменты автоматизации на разных уровнях приложения. В главе также представлен обзор инструментов автоматизированного тестирования на основе искусственного интеллекта и машинного обучения (ИИ/МО), перечислены антипаттерны автоматизированного тестирования и даны советы, как их избежать. Готовы? Поехали!

Введение

Для начала вспомним обсуждение из главы 1, где мы говорили о том, что для создания качественного приложения тестирование необходимо практиковать как на микро-, так и на макроуровне. Это относится и к автоматизированному функциональному тестированию.

Когда дело доходит до реализации такого тестирования, некоторые организации сосредотачиваются исключительно на тестах макроуровня, добавляя все больше и больше сквозных функциональных тестов, управляемых пользовательским интерфейсом, и полностью упускают из виду тесты на микроуровне приложения. Например, у одной команды, которую я консультировала, было более 200 сквозных функциональных тестов на основе пользовательского интерфейса. Каждый день на выполнение пакета тестов уходило восемь часов, и в конце оно заканчивалось неудачей из-за хрупкой природы тестов макроуровня. Это явный антипаттерн, поскольку такой набор тестов не только противоречит главной цели — быстрому получению обратной связи, но и не обеспечивает стабильности последней. Вот почему командам необходимо создавать автоматизированные тесты как на микро-, так и на макроуровне: тесты на микроуровне выполняются быстрее и более стабильны.

Начнем со знакомства с различными типами тестов на микро- и макроуровнях. Позже рассмотрим несколько упражнений, которые помогут вам их реализовать.

Введение в типы микро- и макротестов

Для обсуждения различных типов тестов важно отметить четыре характеристики: масштаб, в котором они работают, цель, которую преследуют, скорость предоставления обратной связи и объем усилий для их создания и поддержки. Понимание этих фундаментальных характеристик позволит вам организовать свои усилия по автоматизированному тестированию проекта (то есть выбрать, какие из них

применять в зависимости от потребностей проекта). Чтобы объяснить различные типы тестов, снова воспользуемся гипотетическим приложением электронной коммерции из главы 2.

Как показано на рис. 3.1, приложение имеет три уровня: пользовательский интерфейс, сервисы RESTful (аутентификации — Auth, обслуживания клиентов — Customer и управления заказами — Order) и базу данных (БД). Пользовательский интерфейс взаимодействует с сервисами, передавая им данные для обработки, а сервисы взаимодействуют с базой данных для сохранения/извлечения соответствующей информации. Приложение также интегрируется с внешним сервисом управления информацией о товарах и нижестоящими системами (системой управления складом и т. д.) для выполнения заказов. Типичный пользовательский сценарий в приложении выглядит так: он вводит в пользовательском интерфейсе свои учетные данные, которые отправляются в сервис аутентификации Auth для проверки, и при успешном входе в систему ищет товары и размещает заказы через пользовательский интерфейс. В обязанности сервиса управления заказами Order входят прием заказов от пользователя, проверка информации о товарах с помощью внешнего сервиса управления информацией о товарах и передача ее в систему управления складом для запуска процесса доставки.

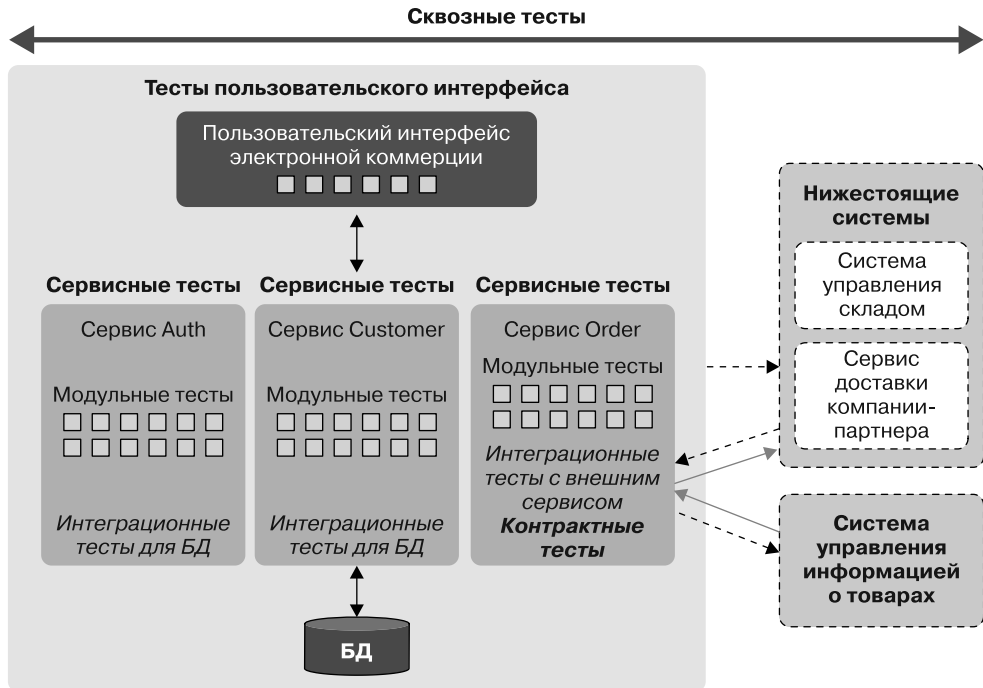


Рис. 3.1. Микро- и макротесты на соответствующих уровнях приложения с сервис-ориентированной архитектурой

На рис. 3.1 показаны различные тесты микро- и макроуровня на соответствующих уровнях для автоматизированного функционального тестирования этого приложения. Рассмотрим их по очереди.

Модульные тесты

Модульные тесты присутствуют во всех сервисах нашего примера приложения, а также на уровне пользовательского интерфейса. Их цель — обеспечить безопасность сети на микроуровне. Они проверяют мельчайшие детали функциональности приложения, например поведение метода в классе. На этом уровне создается большинство автоматических тестов, проверяющих входные данные.

Допустим, у нас есть метод `return_order_total(item_prices)` в сервисе управления заказами `Order`, который возвращает общую сумму заказа. Тогда мы могли бы добавить модульные тесты для проверки следующих аспектов поведения этого метода:

- возвращает общую сумму, если `item_prices` имеет отрицательное значение из-за скидок;
- возвращает общую сумму, если `item_prices` пустой;
- возвращает общую сумму, если `item_prices` содержит недопустимое значение (например, включающее буквы, символы и т. д.);
- возвращает общую сумму, если цены на товары передаются с разными символами валют и разделителями, что необходимо для поддержки локализации приложения;
- возвращает правильно округленную общую сумму с фиксированным числом знаков после запятой.

Модульные тесты проверяют код приложения и создаются разработчиками. В командах, практикующих методику разработки через тестирование (Test-Driven Development, TDD), разработчики пишут модульные тесты до того, как будет написан прикладной код, заставляя их терпеть неудачу, а затем добавляют ровно столько прикладного кода, чтобы эти тесты выполнялись успешно. Такая практика помогает избежать появления избыточной и непроверенной логики в коде. Для создания модульных тестов прикладного кода часто применяются фреймворки JUnit, TestNG и NUnit, а для модульного тестирования фронтенда — Jest, Mocha и Jasmine.

Модульные тесты выполняются быстрее всего. Поскольку они проверяют прикладной код, их легко создавать и поддерживать. Часто они запускаются как часть этапа сборки приложения на локальной машине разработчика, что обеспечивает раннее тестирование, а также быструю и раннюю обратную связь.

Интеграционные тесты

В большинстве средних и крупных веб-приложений существует довольно много точек интеграции с внутренними или внешними компонентами, такими как сервисы, пользовательский интерфейс, базы данных, кэши, файловые системы и т. д., которые могут быть распределены по сети и инфраструктуре. Чтобы проверить работу всех этих точек, необходимо написать интеграционные тесты, которые будут взаимодействовать с реальными интегрируемыми системами. В центре внимания таких тестов должна быть проверка примеров удачной и неудачной интеграции, а не сквозная проверка функциональности. Поэтому в идеале они должны быть такими же маленькими, как модульные тесты.

В примере приложения электронной коммерции сервис управления заказами интегрируется с внутренними компонентами, такими как пользовательский интерфейс и база данных, и с другими сервисами. Он интегрируется также с внешним сервисом управления информацией о товарах и нижестоящими системами. Мы должны написать интеграционные тесты для проверки интеграции с каждым сервисом, чтобы проверить правильность взаимодействия с ними, и оценить необходимость создания интеграционных тестов для проверки интеграции сервиса управления заказами с внешними системами и службами.

Интеграционные тесты можно писать с использованием платформ модульного тестирования, подобных тем, что упоминались в предыдущем разделе, а также специальных инструментов моделирования интеграции. Например, с помощью JUnit и Spring Data JPA (<https://oreil.ly/jeu9l>) можно создавать интеграционные тесты для проверки взаимодействия с базами данных. Эти тесты проверяют также прикладной код, поэтому могут создаваться и поддерживаться разработчиками. Скорость их выполнения зависит от времени ответа внешней системы, поэтому они могут выполняться медленнее, чем модульные тесты, которые действуют в полной изоляции.

Контрактные тесты

Выполнить интеграционное тестирование может оказаться невозможно, если интегрируемые сервисы тоже находятся в стадии разработки. Такое часто случается при создании крупных приложений несколькими командами, которые занимаются разными сервисами. В подобных проектах команды согласовывают стандартный контракт для каждого сервиса и работают с их заглушками, пока не будут готовы сами сервисы. Однако при использовании заглушек вы не будете знать, изменились ли фактические контракты интегрируемых сервисов! Если такое случится, вы продолжите создавать новые функции, опираясь на недействительные контракты, пока не обнаружите, что они недействительные, во время интеграционного тестирования с реальными сервисами в конце цикла разработки. Это одна из основных причин проведения контрактных тестов.

Контрактные тесты написаны для проверки соответствия заглушек фактическим контрактам интегрируемых сервисов и постоянного предоставления обратной связи обеим командам по мере их продвижения в разработке. Контрактные тесты не обязательно проверяют точные данные, возвращаемые интегрируемым сервисом, они скорее фокусируются на самой структуре контракта. В нашем приложении электронной коммерции можно добавить тесты для проверки контракта внешнего сервиса управления информацией о товарах, чтобы при каждом его изменении можно было соответствующим образом изменять функции сервиса управления заказами. Контрактные тесты можно создавать также для проверки контракта взаимодействий между пользовательским интерфейсом и сервисами, если разработка происходит параллельно. Комплексный рабочий процесс контрактного тестирования предполагает сотрудничество между командами и подробно обсуждается далее в этой главе. А автоматизировать этот процесс можно с помощью инструментов Postman и Pact.

Контрактные тесты, как правило, выполняются очень быстро, потому что их объем невелик (они просто проверяют структуру контракта). Они включены в базу кода приложения и, следовательно, могут создаваться и обслуживаться самими разработчиками, хотя они и сложнее модульных тестов. Дополнительная сложность связана с комплексной настройкой, требующей взаимодействия между командами.

Сервисные тесты

Как обсуждалось в главе 2, API следует рассматривать как самостоятельные продукты и тщательно тестировать их независимо от поведения пользовательского интерфейса. Именно это является целью сервисных тестов.

Сервисы фактически реализуют всю логику предметной области, такую как бизнес-правила, критерии ошибок, механизмы повторных попыток, хранение данных и т. д. Они отклоняют недействительные запросы после проверки их структуры и формата значений. И именно с них начинается тестирование на макроуровне, потому что сервисные тесты охватывают интеграцию, рабочие сценарии и т. д. Например, мы могли бы добавить в свое приложение электронной коммерции несколько сервисных тестов для проверки сервиса управления заказами, которые помогают убедиться, что:

- только авторизованный пользователь может создать новый заказ;
- заказ создается, только если выбранные товары доступны на момент его создания;
- для позитивных и негативных входных данных возвращаются правильные коды состояния HTTP.

Аналогично каждый сервис должен иметь тесты для всех своих конечных точек.

Сервисные тесты иногда размещаются в отдельной базе кода, но для быстрого получения обратной связи их лучше хранить вместе с компонентами и сервисами. Создавать и поддерживать эти тесты немного сложнее, чем модульные, потому что они требуют подготовки тестовых данных в БД. Обычно за такие тесты отвечают

особые члены команды — тестировщики. Они выполняются быстрее, чем сквозные тесты, управляемые пользовательским интерфейсом, и немного медленнее, чем предыдущие три вида тестов микроуровня (модульные, интеграционные и контрактные тесты). Для автоматизации тестов API можно применять такие инструменты, как REST Assured, Karate и Postman.



Любая сущность, которую можно независимо повторно использовать или заменить, например сервис, называется компонентом. Услышав термин «компонентные тесты», можете смело интерпретировать его как сервисные тесты.

Функциональные тесты пользовательского интерфейса

Функциональные тесты пользовательского интерфейса выполняются в браузере и имитируют действия пользователя в приложении. Они помогают получить обратную связь об интеграции между несколькими компонентами, такими как сервисы, пользовательский интерфейс и база данных, и должны быть сосредоточены на проверке всех критически важных пользовательских сценариев. Одним из примеров критического пути в приложении электронной коммерции может служить поиск товара, его добавление в корзину, оплата и подтверждение заказа, а проверку можно реализовать в виде функционального теста пользовательского интерфейса. При написании таких тестов избегайте повторной проверки деталей, которые проверяются тестами на микроуровне, потому что это лишь увеличит время выполнения тестов, не давая дополнительных выгод. Например, проверка итоговой суммы заказа для различных комбинаций цен на товары должна выполняться модульными тестами и не требует повторной проверки в рамках функционального тестирования пользовательского интерфейса.

Обычно функциональные тесты пользовательского интерфейса хранятся отдельно от прикладного кода и создаются и обслуживаются тестировщиками, иногда в сотрудничестве с разработчиками. Эти тесты выполняются дольше, и, как правило, они довольно хрупкие, потому что зависят от стабильности поведения всего программного стека, включая инфраструктуру, сеть и т. д. Кроме того, их обслуживание, в отличие от тестов других типов, требует значительных усилий, потому что сбои в приложении могут происходить где угодно, например, может измениться идентификатор элемента, увеличиться задержка загрузки страницы или сервис может оказаться недоступным из-за проблем со средой.

Для автоматизации тестов пользовательского интерфейса широко применяются такие инструменты, как Selenium и Cypress. Далее в этой главе вы найдете упражнения, требующие применения обоих.



Каждый раз, размышляя о добавлении функционального теста пользовательского интерфейса, сначала подумайте о его цели (например, проверка входных данных, бизнес-правил уровня обслуживания и т. д.) и посмотрите, можно ли ее достичь, написав микротесты более низкого уровня.

Сквозные тесты

Как следует из названия, сквозные тесты должны проверять весь рабочий процесс, включая нижележащие системы. В нашем примере после размещения заказа на веб-сайте нижележащие системы (система управления складом, сторонние партнерские службы доставки и т. д.) фактически выполняют заказ. Необходимо протестировать весь этот процесс и проверить правильность интеграции сервисов.

В зависимости от контекста приложения функциональные тесты пользовательского интерфейса часто становятся сквозными. Если у вас это не так, то создайте отдельные сквозные тесты с помощью комбинации инструментов тестирования пользовательского интерфейса, сервисов и БД, чтобы охватить все точки интеграции. Очевидно, что эти тесты будут выполняться дольше всего и потребуют большей осторожности при обслуживании, потому что им нужна стабильная среда и настройка тестовых данных в различных системах. Цель этих тестов — проверить правильность интеграции всех компонентов, а не их функциональные возможности. И вы можете выполнить всего несколько тестов, которые активируют все ваши компоненты.



Общепринятой практикой считается то, что разработчики пишут тесты микроуровня во время разработки, а тестировщики — тесты макроуровня на этапе тестирования. Но такое распределение обязанностей во многом зависит от набора навыков, имеющихся у специалистов, и может варьироваться от команды к команде.

Итак, мы рассмотрели все типы микро- и макротестов, так что вы должны иметь представление об их четырех основных характеристиках. В следующем разделе поговорим о стратегии автоматизированного функционального тестирования, широко используемой командами разработчиков ПО. Можете взять ее как основу для определения стратегии, соответствующей конкретным потребностям вашего проекта.

Стратегия автоматизированного функционального тестирования

Если говорить кратко, то стратегию автоматизированного тестирования можно выразить одним предложением: *добавьте тесты для проверки достаточно полного объема функциональности на нужных уровнях приложения, чтобы они давали максимально быструю обратную связь команде*. Майк Кон прекрасно сформулировал это в своей книге *Succeeding with Agile*¹ (Addison-Wesley Professional) в виде

¹ Кон М. Scrum: гибкая разработка ПО. Описание процесса успешной гибкой разработки ПО с использованием Scrum. — М., 2017.

пирамиды тестирования. Согласно этой идее рекомендуется создавать широкий набор тестов на микроуровне и постепенно сокращать их количество на макроуровне по мере увеличения их объема. Например, если у вас есть $10x$ модульных и интеграционных тестов, то должно быть $5x$ сервисных тестов и только x тестов, управляемых пользовательским интерфейсом. Если представить их в виде стопки, расположив в самом низу модульные тесты, то они образуют пирамиду. Очевидная причина такой рекомендации заключается в том, что с увеличением объема тестов на их выполнение уходит больше времени, а их написание и обслуживание обходятся дороже.



Помимо пирамиды, существуют и другие способы представления автоматизированных тестов, например в виде сот или наградного кубка (<https://oreil.ly/IMadd>). По сути, все они подчеркивают один и тот же принцип: тесты на микроуровне легче писать и запускать, чем тесты на макроуровне. Если вы решите исследовать эти формы представления наборов тестов, то обратите внимание на то, как они определяют область действия каждого типа тестов — они меняются по мере изменения объема тестирования.

Типичная пирамида тестирования для сервис-ориентированного веб-приложения, такого как в нашем примере, показана на рис. 3.2.



Рис. 3.2. Пирамида тестирования для сервис-ориентированного веб-приложения

Как и многим специалистам по тестированию, мне доводилось видеть работу пирамиды тестирования на практике. Могу привести один примечательный пример: реорганизовав проект, в котором было более 200 сквозных тестов пользовательского интерфейса, чтобы привести его в соответствие с пирамидой тестирования, мы стали получать обратную связь в течение 35 минут после отправки изменений кода в репозиторий, при этом к коду применялось ~470 тестов!



Пирамиду тестирования можно рассматривать как идеал, к которому следует стремиться, но на практике не всегда получается достичь формы пирамиды. Это может быть связано со следующими недостатками: отсутствием полноценной тестовой среды для поддержки сквозных тестов или инструментов автоматизации некоторых видов функций, таких как сканирование штрихкода, или, говоря прямо, отсутствием навыков. В таких случаях команде следует помнить о компромиссах, на которые она идет, и выбирать количество и типы тестов так, чтобы они помогали получать быструю обратную связь, несмотря на эти ограничения.

Другая часть стратегии автоматизации — наличие возможности отслеживать долю кода, охваченного автоматизированными тестами, чтобы гарантировать отсутствие белых пятен. Для этой цели можно использовать инструменты управления тестированием, такие как TestRail, инструменты управления проектами, такие как Jira, или что-то более простое, например лист Excel. Отслеживание охвата автоматизированными тестами имеет большое значение. По разным причинам многие команды откладывают автоматизацию тестирования пользовательской истории или вообще отказываются от нее, что приводит к задержке и получению неполной обратной связи. В результате разработчики могут утратить доверие к самому пакету автоматизации. Выявление всех тестовых сценариев и их автоматизация помогают избежать этого. Идеальная практика, которой придерживаются многие Agile-команды, — называть пользовательскую историю реализованной, только если все ее тесты на микро- и макроуровне автоматизированы!

Упражнения

После знакомства с типами тестов настала пора окунуться в код. Упражнения, приведенные в этом разделе, помогут вам начать настройку среды функционального тестирования на трех уровнях приложения: я покажу, как реализовать функциональные тесты на основе пользовательского интерфейса с помощью Selenium и Cypress, сервисные тесты с помощью REST Assured и модульные тесты с применением JUnit. Поехали!

ТЕХНИЧЕСКИЙ СТЕК АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ

Вот несколько советов, которые следует учитывать при выборе стека технологий автоматизации.

- Лучше всего, чтобы он имел как можно больше общего со стеком разработки, тогда членам команды не придется изучать совершенно новый набор инструментов. Я заметила: когда команды используют разные наборы технологий для разработки и тестирования, у разработчиков возникает естественное сопротивление созданию тестов, что препятствует достижению цели раннего тестирования и обеспечению более быстрого получения обратной связи.
- Не старайтесь собрать тесты со всех уровней в общую базу кода, храните их в соответствующих компонентах, чтобы они переносились вместе с компонентами при повторном применении. Такой подход означает, что выбор технологического стека на каждом уровне будет зависеть от технологического стека разработки соответствующего компонента.

Функциональные тесты пользовательского интерфейса

Для создания автоматизированных функциональных тестов пользовательского интерфейса часто задействуются два инструмента: Selenium WebDriver и Cypress. Selenium WebDriver позволяет писать тесты на многих языках программирования, таких как Java, C#, Python, JavaScript и т. д. Cypress, напротив, поддерживает только JavaScript, зато этот инструмент обладает множеством преимуществ. Я покажу, как использовать оба, поэтому вы сможете выбрать подходящий инструмент, исходя из предпочтений вашей команды и применяемого набора технологий разработки.

Фреймворк Selenium WebDriver для Java

Вначале создадим основу для автоматизированных тестов с помощью Java и Selenium WebDriver (<https://www.selenium.dev/projects>).

Предварительные условия. Прежде чем продолжить, нужно установить следующие инструменты:

- последнюю версию Java (<https://oreil.ly/eT0qE>);
- интегрированную среду разработки (IDE) по своему выбору. У разработчиков на Java большой популярностью пользуется, например, IntelliJ (<https://oreil.ly/2950c>);
- браузер Chrome (https://www.google.com/intl/en_in/chrome).

Потребуется установить и несколько других инструментов, как описано в следующих разделах.

Maven. Apache Maven — это инструмент автоматизации сборки. Инструменты автоматизации сборки помогают стандартизировать процессы управления зависимостями и этапы сборки проекта. Например, во многих проектах для реализации новых функций используются сторонние библиотеки и плагины, и зачастую крайне важно, чтобы все члены команды брали одни и те же версии библиотек и плагинов и выполняли одну и ту же последовательность шагов для создания артефактов приложения (то есть собирали проект, запускали тесты и т. д.). Инструменты автоматизации сборки помогают достичь этих целей. Разработчикам на Java наиболее хорошо известны два инструмента: Maven и Gradle. В этом упражнении мы используем Maven, загрузите его (<https://oreil.ly/IjplR>) и следуйте инструкциям по установке на сайте.

Чтобы получить общее представление о том, как работает Maven, взглянем на XML-файл объектной модели проекта (Project Object Model, POM) `pom.xml`. Здесь определяются все зависимые библиотеки, плагины и их версии, как показано в примере 3.1, а Maven заботится обо всем остальном.

Пример 3.1. Пример файла `pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>SeleniumJavaExample</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>15</maven.compiler.source>
    <maven.compiler.target>15</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-java</artifactId>
      <version>4.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>7.4.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Вот важные детали этого файла, на которые следует обратить внимание.

- Атрибуты `groupId`, `artifactId` и `version` определяют характеристики проекта, которые позволят Maven следить за ним в дальнейшем. Эти сведения можно добавить при создании нового проекта в IntelliJ.
- Раздел `properties` определяет, какую версию Java использовать для компиляции. Сюда можно включать также переменные, специфические для конкретных проектов, которые будут применяться в других разделах в файле `pom.xml` или в коде приложения.
- Раздел `dependencies` содержит список зависимых библиотек и их версий. Обратите внимание на то, что файл ссылается на Selenium 4.0 как на зависимость. Maven поддерживает центральное хранилище для всех библиотек и их версий, откуда загружает их на ваш компьютер, руководствуясь настройками, содержащимися в этом разделе файла `pom.xml`. Такой централизованный способ управления библиотеками гарантирует, что все члены команды будут иметь одинаковые двоичные файлы всех библиотек. Чтобы добавить зависимость для библиотеки в файл `pom.xml`, найдите библиотеку в репозитории Maven (<https://oreil.ly/lMhEf>) и скопируйте ее характеристики в раздел `dependencies`.

Аналогично можно настроить плагины, среды и т. д., объявив соответствующие атрибуты в файле `pom.xml`, как описано в документе Maven (<https://oreil.ly/eCamH>). Далее мы используем файл из примера 3.1 для разработки тестов Selenium WebDriver.

Maven также предоставляет команды управления жизненным циклом сборки и создания артефактов приложения. Вот наиболее часто применяемые из них:

`mvn compile`

Компилирует код проекта.

`mvn clean`

Очищает (то есть удаляет) артефакты, созданные ранее.

`mvn test`

Запускает тесты, написанные с использованием фреймворка тестирования (один из них мы настроим далее).

Существуют также команды Maven для установки, развертывания и выполнения других действий в конце жизненного цикла создания артефактов приложения.

TestNG. TestNG — это фреймворк тестирования, подобный Junit — другому фреймворку тестирования, популярному у разработчиков на Java. Фреймворки тестирования, как правило, дают возможность создавать тесты, добавлять утверждения, определять задачи настройки перед тестированием и освобождения ресурсов

после него, организовывать тесты в группы, выполнять тесты и представлять сводную информацию с результатами их выполнения. TestNG можно применять для разработки всех типов тестов: модульных, интеграционных и сквозных. Чтобы установить его, достаточно добавить зависимость в файл `pom.xml`, как показано в примере 3.1.

Вот несколько важных особенностей TestNG, которые можно использовать регулярно.

- **@Test** — аннотация, определяющая метод в классе как тестовый метод, выполняемый под управлением TestNG. Она должна предшествовать каждому тесту.
- **@BeforeClass**, **@AfterClass**, **@BeforeMethod**, **@AfterMethod**, **@BeforeSuite**, **@AfterSuite**. Как следует из названий, методы, отмеченные этими аннотациями, запускаются до или после тестовых классов, методов или всего набора тестов. Этими тегами можно отмечать методы, подготавливающие тестовую среду и освобождающие ресурсы после тестов.
- **assertEquals()**, **assertTrue()** и другие методы **assert*** используются для выполнения проверок в тестах. IntelliJ поможет вам разобраться в синтаксисе этих методов.

Selenium WebDriver. Selenium, популярный инструмент с открытым исходным кодом для автоматизации тестирования, был создан Джейсоном Хаггинсом в 2004 году. С тех пор он претерпел множество воплощений. Познакомиться с увлекательной историей развития этого инструмента можно на его веб-сайте (<https://www.selenium.dev/history>). В настоящее время он поддерживается чрезвычайно активным сообществом.

ПОЧЕМУ SELENIUM ПОЛУЧИЛ ТАКОЕ НАЗВАНИЕ

Selenium (селен) — это химический элемент, используемый как противоядие при отравлении ртутью. До появления Selenium самым популярным инструментом автоматизированного тестирования был Mercury (ртуть). Уловили шутку?

Основная цель Selenium WebDriver — упростить взаимодействие с веб-приложением, отображаемым в браузере. Он не предоставляет инструкций для проверки условий, не генерирует отчеты и т. д., поэтому, чтобы получить готовую среду тестирования, нам понадобятся другие инструменты, такие как TestNG и Maven.

Selenium WebDriver состоит из трех основных компонентов.

API

Предлагает методы для взаимодействий с элементами приложения в браузере (кнопки, поля ввода и т. д.).

Клиентская библиотека

Объединяет API, которые можно использовать в наборе тестов. Клиентские библиотеки доступны на многих языках программирования.

Драйвер

Это компонент, фактически выполняющий в браузере действия, инициируемые с помощью API. Драйверы обычно создаются и поддерживаются самими браузерами и не являются частью дистрибутива Selenium. Например, для запуска тестов в Chrome необходимо загрузить ChromeDriver отдельно и включить его в свои сценарии автоматизации.

Для начала познакомимся с различными API, предоставляемыми Selenium WebDriver. В примере 3.2 перечислены некоторые методы WebDriver, часто используемые для поиска различных элементов в приложении. Selenium идентифицирует элементы на веб-странице по значениям их HTML-атрибутов, таких как `id`, `className`, `cssSelector` и т. д. Чтобы получить эти значения, можно выбрать в контекстном меню пункт *Inspect* (Просмотреть код). Попробуйте, например, отыскать текстовое поле поиска Amazon, и вы обнаружите, что он имеет идентификатор `"twotabsearch textbox"`.

Пример 3.2. Некоторые часто используемые методы WebDriver для поиска элементов

```
// найти элемент по атрибуту id
driver.findElement(By.id("login"))

// найти элемент по селектору CSS
driver.findElement(By.cssSelector("#login"));

// найти элемент по имени класса
driver.findElement(By.className("login-card"));

// найти элемент по пути XPath
driver.findElement(By.XPath("//@login"));

// найти несколько элементов
driver.findElements(By.cssSelector("#username li"));
```



Атрибуты `id` элементов имеют уникальные значения в пределах страницы. Следовательно, это предпочтительный признак для поиска, так как он обеспечивает стабильность результатов тестирования. Селекторы CSS и пути XPath могут терять свою актуальность, когда приложение часто меняется.

Selenium WebDriver также предоставляет дополнительные средства поиска элементов относительно других элементов (<https://oreil.ly/eWukW>), например указывая, что они находятся выше (*above*), ниже (*below*) или левее (*toLeftOf*) другого элемента.

Получив искомый элемент, вы можете взаимодействовать с ним. В примере 3.3 перечислены несколько часто применяемых методов Selenium WebDriver для выполнения различных действий с элементами.

Пример 3.3. Некоторые часто используемые методы WebDriver для взаимодействия с веб-элементами

```
// щелчок на элементе
driver.findElement(By.id("submit")).click();

// ввод текста в поле ввода
driver.findElement(By.cssSelector("#username")).sendKeys(username);
```

Для более сложных взаимодействий, таких как `keyDown`, `contextClick` и `dragAndDrop`, можно использовать класс `Actions` (<https://oreil.ly/wV81u>).

Помимо методов взаимодействий, WebDriver предоставляет также методы управления поведением браузера, такие как открытие URL, возврат назад, закрытие браузера, настройка размера окна браузера, установка cookie-файлов в браузере, переключение между несколькими вкладками и т. д. В примере 3.4 показаны некоторые методы, часто применяемые для манипуляций браузером.

Пример 3.4. Некоторые методы WebDriver, часто используемые для управления поведением браузера

```
// открыть URL
driver.get("https://example.com");

// кнопки "Назад", "Вперед" и "Обновить"
driver.navigate().back();
driver.navigate().forward();
driver.navigate().refresh();

// открыть окно браузера с размером, соответствующим размеру экрана iPad
driver.manage().window().setSize(new Dimension(768, 1024));

// закрыть окно браузера
driver.close();

// закрыть сеанс работы с драйвером
driver.quit();
```

При навигации по страницам тест должен дожидаться завершения загрузки страницы или появления элемента. Некоторые тестировщики используют для этого жестко запрограммированные операторы задержки, но такой подход делает тесты хрупкими, потому что в разных средах время загрузки страниц может меняться. WebDriver предлагает несколько готовых стратегий ожидания для решения этой проблемы.

- Стратегия *неявного* ожидания заставляет WebDriver опрашивать объектную модель документа (Document Object Model, DOM), представляющую все со-

держимое HTML-документа, в течение x секунд, ожидая появления элемента. По умолчанию WebDriver ждет 0 секунд, но можно изменить это время на этапе инициализации драйвера и установить типичное время ожидания.

- Стратегия *явного* ожидания заставляет WebDriver ждать до x секунд, прежде чем ожидаемое условие станет истинным.
- Стратегия *активного* ожидания обеспечивает бóльшую гибкость. Она заставляет WebDriver ждать не более x секунд, пока ожидаемое условие станет истинным, проверяя его выполнение каждые y секунд.

Эти стратегии ожидания показаны в примере 3.5.

Пример 3.5. Стратегии ожидания в WebDriver

```
// Неявное ожидание в течение 10 с,  
// прежде чем истечет тайм-аут  
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));  
  
// Явное ожидание в течение 10 с,  
// прежде чем кнопка отправки формы станет активной  
WebElement submitButton = new WebDriverWait(driver, Duration.ofSeconds(10)).  
    until(ExpectedConditions.elementToBeClickable(By.id("submit")));  
  
// Активное ожидание, когда опрос осуществляется  
// каждую секунду, но не дольше 3 с,  
// пока индикатор прогресса не исчезнет  
FluentWait wait = new FluentWait(driver)  
    .withTimeout(Duration.ofSeconds(3))  
    .pollingEvery(Duration.ofSeconds(1))  
    .ignoring(NoSuchElementException.class);  
wait.until(ExpectedConditions.invisibilityOf(driver.findElement(By.  
id("spinner"))));
```

Эти методы WebDriver часто используются в повседневной практике тестирования. WebDriver предлагает также множество более продвинутых взаимодействий, таких как прослушивание событий и выполнение различных действий в зависимости от типа события, взаимодействие с модальными окнами и почти все остальное, что может понадобиться протестировать в браузере. Selenium 4 также позволяет имитировать ответы сервера и выполнять отладку с помощью протокола Chrome DevTools (<https://oreil.ly/D8Fkb>). За более подробной информацией об этих расширенных возможностях обращайтесь на веб-сайт проекта (<https://oreil.ly/WdovT>).

Page Object Model. Page Object Model — это шаблон проектирования, часто применяемый для организации среды автоматизации на основе пользовательского интерфейса. Он предполагает воссоздание структуры приложения такой, какая она есть в среде автоматизации, то есть создание класса для каждой страницы вашего приложения и определение элементов и действий на странице в этом классе. Это довольно эффективный шаблон, потому что допускает абстракцию и инкапсуляцию и, следовательно, упрощает исправление проблем или добавление новых из-

менений. Например, при изменении идентификатора элемента вы знаете, как его отыскать (в классе страницы) и исправить. Если у вас нет такой абстракции, вам придется явно изменить идентификатор во всех тестах.

В примере 3.6 показан класс `LoginPage` с тремя элементами: полями ввода имени пользователя и пароля и кнопкой входа. Он также имеет метод `login(email, password)` для выполнения входа. Мы будем обращаться к этому классу `LoginPage` позже, при создании теста.

Пример 3.6. Класс `LoginPage` с использованием Page Object Model

```
// LoginPage.java
```

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class LoginPage {

    private WebDriver driver;
    private By emailID = By.id("user_email");
    private By passwordField = By.id("user_password");
    private By signInButton =
        By.cssSelector("input.gr-button.gr-button--large");

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public HomePage login(String email, String password){
        driver.findElement(emailID).sendKeys(email);
        driver.findElement(passwordField).sendKeys(password);
        driver.findElement(signInButton).click();
        return new HomePage(driver);
    }
}
```

Аналогично фреймворк автоматизации должен поддерживать классы, представляющие все страницы вашего приложения.

Настройка и рабочий процесс. Следующий шаг после исследования всех компонентов, необходимых для организации типичной среды Java/Selenium WebDriver автоматизированного тестирования пользовательского интерфейса, — собрать их вместе и написать тест для простого сценария, предусматривающего вход в приложение (выберите приложение, в котором у вас есть аккаунт) и проверку заголовка главной страницы. Чтобы создать такой тест, выполните следующие действия.

1. Откройте IntelliJ и создайте новый проект Maven, выбрав в меню пункт **File** ▶ **New** ▶ **Project** ▶ **Maven** (Файл ▶ Создать ▶ Проект ▶ Maven).

2. Выберите версию Java, которую вы используете.
3. Перейдите к следующему окну и введите название проекта, местоположение, идентификатор группы `groupId` и идентификатор артефакта `artifactId` (рис. 3.3).

Рис. 3.3. Создание нового проекта Maven в IntelliJ

Выполнив эти три шага, вы создадите первоначальную структуру проекта, как показано в примере 3.7.

Пример 3.7. Первоначальная структура проекта Maven

```

├─ SeleniumJavaExample.iml
├─ pom.xml
├─ src
│   ├── main
│   │   ├── Java
│   │   └── resources
│   └── test
│       └── Java

```

4. Загрузите исполняемый файл `ChromeDriver` (<https://oreil.ly/g8gP8>), совместимый с вашей локальной версией браузера Chrome. Узнать версию Chrome можно, выбрав в меню пункт `Chrome` ► `About Chrome` (Справка ► О браузере Google Chrome).
5. Поместите исполняемый файл в папку `src/main/resources` внутри вашего проекта.
6. Добавьте зависимости проекта, Selenium, Java и библиотеку TestNG, как показано в примере 3.1. В IntelliJ панель Maven находится справа, ее можно использовать для немедленного обновления и загрузки библиотек.
7. Создайте пакет с именем `base` в `src/test/java`.
8. Добавьте новый файл класса с именем `BaseTests.java`, в котором будут определяться настройки `WebDriver`, как показано в примере 3.8.

Пример 3.8. Класс BaseTests

```
// BaseTests.java

package base;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;

import java.time.Duration;

public class BaseTests {

    protected WebDriver driver;

    @BeforeMethod
    public void setUp(){
        System.setProperty("webdriver.chrome.driver",
            "src/main/resources/chromedriver");
        driver = new ChromeDriver();
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
        driver.get("http://eCommerce.com/sign_in");
    }

    @AfterMethod
    public void teardown(){
        driver.quit();
    }
}
```

Метод `setUp()` выполняет несколько действий: определяет путь к выполняемому файлу `ChromeDriver`, создает экземпляр объекта `ChromeDriver`, задает продолжительность неявного ожидания равной 10 с и открывает URL приложения с помощью объекта `driver`. Метод `tearDown()` решает только одну задачу — завершает сеанс браузера после запуска теста. Обратите внимание на аннотации `@BeforeMethod` и `@AfterMethod`. Это аннотации TestNG, они используются для создания и завершения нового сеанса драйвера, который будет запускаться для каждого теста.

9. Затем создайте новый пакет с именем `tests` в `src/test/java` и добавьте первый тестовый класс, например `LoginTest`, как показано в примере 3.9. Обратите внимание на аннотацию `@Test` и метод `AssertEquals()`, предоставляемые фреймворком TestNG.

Пример 3.9. Класс LoginTest с тестом

```
// LoginTest.java

package tests;

import base.BaseTests;
import org.testng.annotations.Test;
```

```
import pages.LoginPage;
import static org.testng.Assert.*;

public class LoginTest extends BaseTests {

    @Test
    public void verifySuccessfulLogin(){
        LoginPage loginPage = new LoginPage(driver);
        assertEquals(loginPage.login("example@gmail.com",
            "Admin123").getTitle(), "Home page");
    }
}
```

10. После написания тестов нужно создать классы страниц. Создайте новый пакет с именем `Pages` в `src/main/java` и добавьте в него свои классы страниц. (Примеры класса `LoginPage` и класса `HomePage` можно увидеть в примерах 3.6 и 3.10 соответственно.)

Пример 3.10. Класс `HomePage`

```
// HomePage.java

package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import java.time.Duration;

public class HomePage {

    private WebDriver driver;
    private By searchField = By.cssSelector("input.searchBox");

    public HomePage(WebDriver driver) {
        this.driver = driver;
    }

    public String getTitle(){
        WebDriverWait wait = new WebDriverWait(driver,
            Duration.ofSeconds(10));
        wait.until(ExpectedConditions.
            presenceOfElementLocated(searchField));
        return driver.getTitle();
    }
}
```

Классы страниц будут вызывать обсуждавшиеся ранее методы Selenium `WebDriver` для поиска элементов в этих классах и взаимодействий с ними. Обратите внимание на то, что классы страниц реализуют цепочечный интерфейс, возвращая объекты других страниц. Например, метод `login()` в классе `LoginPage`

возвращает объект `HomePage` вместе с объектом `driver`. Также помните, что методы-утверждения не принадлежат классам страниц!

11. Теперь можно запустить тест из самой IDE, щелкнув правой кнопкой мыши на зеленом треугольнике рядом с аннотацией `@Test` или выполнив команду Maven следующим образом:

```
$ mvn clean test
```

Эта команда откроет браузер Chrome и запустит тест. Она также создаст отчет в формате HTML в `target/surefire-reports/index.html` (рис. 3.4).

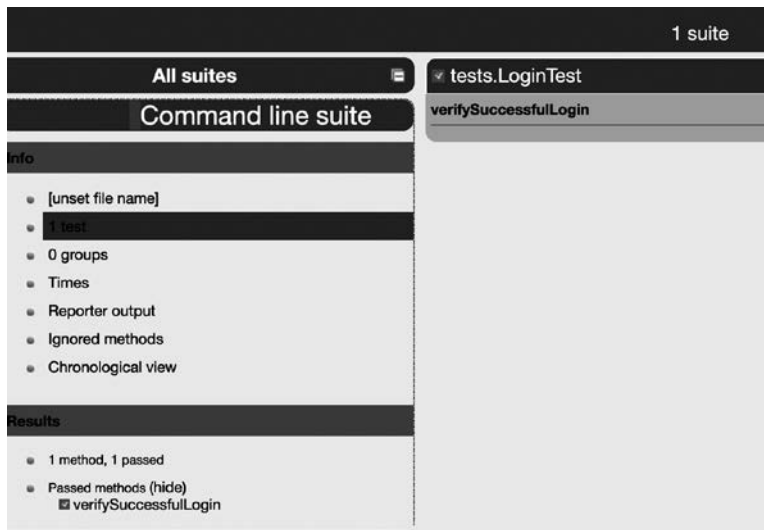


Рис. 3.4. Отчет в формате HTML, сгенерированный плагином Maven Surefire

Поздравляю, вы успешно написали и запустили свой первый тест!

Этот тест выполнялся успешно, но в случае сбоев в конвейере CI вам могут очень пригодиться скриншоты. Чтобы организовать создание скриншотов при сбоях, измените метод `teardown()`, как показано в примере 3.11. Создайте папку с именем `screenshots` в `src/main/resources`, и скриншоты с сообщениями об ошибках будут помещаться туда.

Пример 3.11. Создание скриншотов при сбоях

```
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.testng.ITestResult;
import java.io.File;
import java.io.IOException;
import com.google.common.io.Files;
```

```

@AfterMethod
public void teardown(ITestResult result){
    if(ITestResult.FAILURE == result.getStatus()) {
        var camera = (TakesScreenshot) driver;
        File screenshot = camera.getScreenshotAs(OutputType.FILE);
        try {
            Files.move(screenshot,
                new File("src/main/resources/screenshots/" +
                    result.getName() + ".png"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    driver.quit();
}

```

При желании в среду автоматизации можно добавить дополнительные возможности в зависимости от потребностей конкретного проекта. Например, можно запускать тесты параллельно, используя возможности TestNG (<https://oreil.ly/0wME3>) или Selenium Grid (<https://oreil.ly/34bZH>), группировать тесты и запускать их в нескольких браузерах с помощью TestNG (<https://oreil.ly/4tFSc>), подключать фреймворки разработки через поведение (behavior-driven development, BDD), такие как Cucumber (<https://oreil.ly/EtGeG>), и т. д. Однако всегда помните, что количество тестов пользовательского интерфейса должно быть минимальным.

РАЗРАБОТКА ЧЕРЕЗ ПОВЕДЕНИЕ

BDD — это практика разработки программного обеспечения, целью которой является сближение бизнеса и разработчиков. Например, BDD-фреймворки, такие как Cucumber, позволяют писать тесты на естественном языке, напоминающие типичную пользовательскую историю со структурой «Дано, когда, тогда» (<https://oreil.ly/cGGrb>). Это дает возможность представителям бизнеса передавать требования в форме тестов, терпящих неудачу, а техническим специалистам — создавать функции для их успешного преодоления.

Фреймворк JavaScript — Cypress

Фреймворк Cypress (<https://www.cypress.io>) был выпущен в 2014 году, через десять лет после появления Selenium, и получил широкое распространение как комплексный инструмент автоматизации тестов пользовательского интерфейса. В отличие от Selenium, Cypress позволяет писать тесты только на JavaScript. Несмотря на это ограничение, он завоевал большую популярность благодаря следующим важным особенностям.

- Cypress выполняет команды не по сети, как это делает Selenium, а в том же цикле выполнения, что и приложение. Это делает его намного быстрее.

- Cypress включает все инструменты, необходимые для создания сквозных автоматизированных тестов пользовательского интерфейса, и тем самым избавляет от необходимости настраивать дополнительные инструменты, такие как TestNG, Cucumber и т. д. Он позволяет также применять существующие, проверенные инструменты для выполнения определенных задач. Например, Cypress по умолчанию использует Mocha в качестве фреймворка тестирования и Chai — для проверки утверждений.
- Поскольку Cypress встраивается в приложение, он позволяет создавать различные тестовые сценарии, вставлять заглушки прикладных функций, моделировать случаи недоступности сервера путем изменения запросов, настраивать предопределенные состояния приложения и делать многое другое.
- Cypress устраняет нестабильность тестирования из-за неправильно выбранной стратегии ожидания, автоматически ожидая загрузки страницы и доступности ее элементов.
- Cypress значительно упрощает отладку ошибок тестирования, предоставляя скриншоты, логи и видео для каждой команды, выполненной тестом. Эта его особенность позволяет проверять ошибки на странице приложения в их предопределенном состоянии в рамках потока тестирования с помощью Chrome DevTools.

Cypress имеет хорошую поддержку сообщества, которое часто добавляет новые плагины для удовлетворения разнообразных требований. Итак, давайте посмотрим, как настроить автоматизированное тестирование пользовательского интерфейса с помощью Cypress и объектной модели страницы.



Сообщество Cypress выступает за использование паттерна Application Actions Model вместо паттерна Page Object Model. Чтобы узнать об этом больше, прочтите статью в блоге Глеба Бахмутова (<https://oreil.ly/OrhMC>).

Предварительные условия. Для настройки среды автоматизации на JavaScript необходимы следующие инструменты:

- Node.js версии 12 или выше (<https://nodejs.org/en/download>);
- IDE по вашему выбору. Большой популярностью при разработке проектов на JavaScript пользуется Visual Studio Code (<https://oreil.ly/gc3Jn>);
- браузер — Cypress может работать с Chrome, Chromium, Edge, Electron и Firefox.

Cypress. После установки необходимых компонентов выполните следующие пять шагов, чтобы ознакомиться с возможностями Cypress.

1. Создайте каталог проекта. Установите Cypress, выполнив в терминале следующую команду из папки проекта:

```
$ npm install cypress --save-dev
```

- Создайте в этой папке файл `package.json` и добавьте в него код, показанный в примере 3.12.

Пример 3.12. Файл `package.json`

```
{
  "name": "functional-tests",
  "version": "1.0.0",
  "description": "UI Driven End-to-End Tests",
  "main": "index.js",
  "devDependencies": {
    "cypress": "^9.2.0"
  },
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

- Запустите команду:

```
$ node_modules/.bin/cypress open
```

которая откроет приложение Cypress, как показано на рис. 3.5, и настроит структуру среды автоматизации с примерами тестов Cypress для образцового веб-приложения Todo (<https://oreil.ly/8QK2L>).

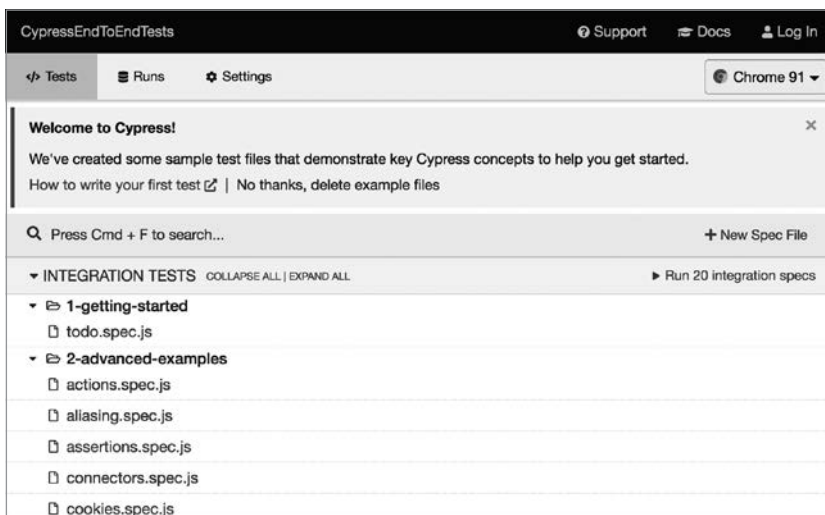


Рис. 3.5. Приложение Cypress с файлами тестов

- После завершения настройки можно попробовать запустить существующие тесты, чтобы получить представление о том, как легко работать с Cypress, а затем

взяться за настройку объектов страниц для конкретного приложения. Выберите предпочитаемый браузер из раскрывающегося списка в правом верхнем углу приложения Cypress и щелкните на любом тестовом файле (`.spec.js`). Cypress откроет браузер, запустит тесты внутри файла и покажет вам отчет (рис. 3.6).

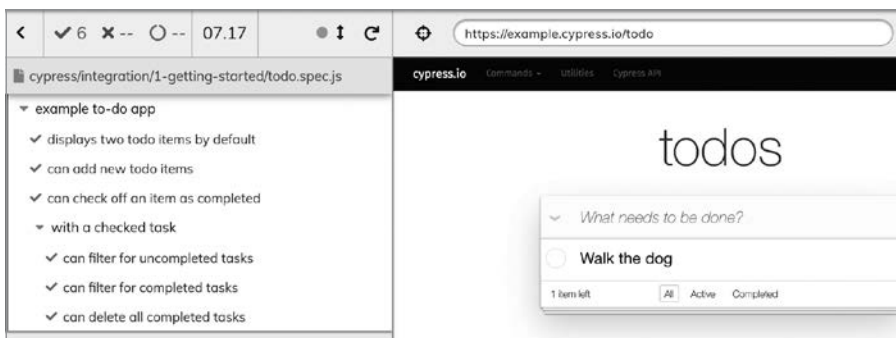


Рис. 3.6. Отчет Cypress о выполнении тестов

Для дальнейшего изучения щелкните на одном из тестов. Вы увидите список выполненных им команд, а при наведении указателя мыши на каждую из команд справа будет показано состояние приложения на момент ее выполнения (рис. 3.7). Что еще можно было бы пожелать для отладки?

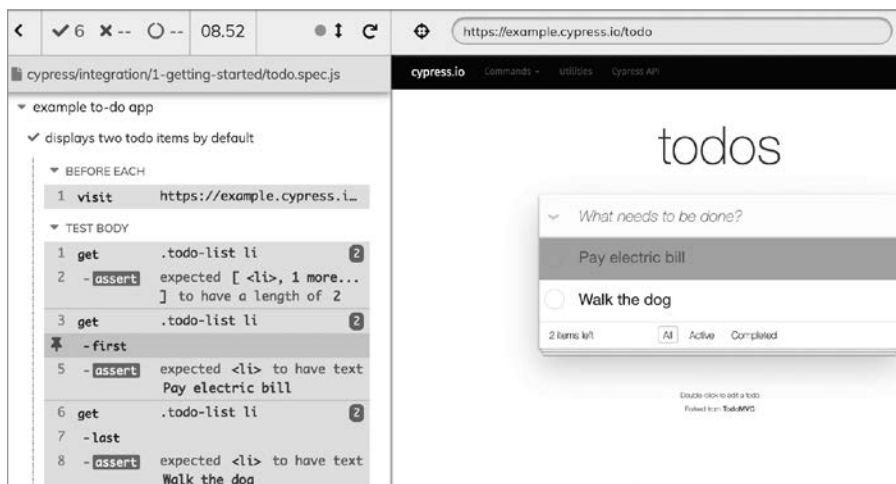


Рис. 3.7. Отладка в Cypress

- Чтобы получить возможность запускать тесты из командной строки, добавьте в файл `package.json` следующий код:


```
"scripts": {
  "test": "cypress run"
}
```

и запустите тест с помощью команды `npm test`. Тесты будут запущены в автономном режиме, а в каталоге проекта появится папка `videos` с видеороликами, иллюстрирующими процесс тестирования.

Теперь, познакомившись с работой Cypress, перечислю методы, которые он предоставляет для взаимодействия с веб-приложением и навигации по нему. Откройте любой тестовый файл — в нем вы увидите некоторые часто используемые методы, например:

- `get(element_locator)` получает веб-элемент из DOM после автоматического ожидания его доступности. Приложение Cypress имеет инструмент для определения локаторов элементов (рис. 3.8). Вы можете применять его при создании объектов страниц;



Рис. 3.8. Определение локатора элементов с помощью Cypress

- `get(element_locator).click()` выполняет щелчок на выбранном элементе;
- `title()` возвращает заголовок страницы;
- `get(select_locator).select(option)` выбирает вариант `option` в раскрывающемся списке;
- `get(element_locator).rightclick()` выполняет щелчок правой кнопкой мыши на выбранном элементе.

Другие методы можно найти в подробной документации инструмента (<https://oreil.ly/6ewls>).

Настройка и рабочий процесс. Для создания среды автоматизации с использованием Cypress и объектной модели страницы требуется всего несколько шагов. Давайте создадим тот же тест, что и в разделе с описанием Selenium, который открывает приложение, выполняет вход в систему и проверяет заголовок домашней страницы. Для этого выполните следующие действия.

1. Создайте новую папку для тестов в `cypress/integration`, например, с именем `ecommerce-e2e-tests`, а в ней — тестовый файл, допустим, `login_tests.spec.js`.
2. Затем создайте новую папку `/page-objects` за пределами папки `/integration` и внутри нее — модули страниц: `login-page.js` и `home-page.js`, как показано в примере 3.13.
3. Созданный тест можно запустить либо непосредственно из приложения Cypress, либо с помощью команды `npm test`.

Пример 3.13. Структура объекта страницы Cypress

```
// page-objects/login-page.js

/// <reference types="cypress" />

export class LoginPage {

  login(email, password){
    cy.get('[id=user_email]').type(email)
    cy.get('[id=user_password]').type(password)
    cy.get('.submitPara > .gr-button').click()
  }
}

// page-objects/home-page.js

/// <reference types="cypress" />

export class HomePage {

  getTitle(){
    return cy.title()
  }
}

// integration/eCommerce-e2e-tests/login_tests.spec.js

/// <reference types="cypress" />

import {LoginPage} from '../../page-objects/login-page'
import {HomePage} from '../../page-objects/home-page'
```

```
describe('example to-do app', () => {
  const loginPage = new LoginPage()
  const homePage = new HomePage()

  beforeEach(() => {
    cy.visit('https://example.com')
  })

  it('should log in and land on home page', () => {
    loginPage.login('example@gmail.com', 'Admin123')
    homePage.getTitle().should('have.string', 'Home Page')
  })
})
```

Обратите внимание на метод `beforeEach()`, предоставляемый фреймворком тестирования Mocha (аналог `@beforeMethod` в TestNG) и открывающий URL приложения перед запуском каждого теста, а также на метод `should('have.string', string)` фреймворка Chai. Оба включены в состав Cypress по умолчанию.

Cypress автоматически запускает тесты после сохранения любых изменений в них. Это упрощает разработку тестов, поскольку позволяет быстро проверить, работает ли новый код должным образом. В главе 7 вы также увидите, как проводить визуальное тестирование с помощью Cypress. В заключение отмечу, что если вы сможете преодолеть барьер изучения JavaScript (что не так уж и сложно), то Cypress принесет вам большую пользу.

Сервисные тесты

Теперь перейдем к сервисным тестам. В этом разделе настроим среду автоматизированного тестирования с помощью библиотеки REST Assured Java для проверки примера REST API. Если вы новичок в API, то ознакомьтесь с вводной информацией в подразделе «Тестирование API» в главе 2.

Предварительные условия

Прежде всего установите следующие компоненты:

- последнюю версию Java (<https://oreil.ly/Uq5Wk>);
- IDE по своему выбору — обычно разработчики на Java выбирают IntelliJ (<https://oreil.ly/y90qz>);
- Maven (<https://oreil.ly/FAOuB>).

Фреймворк Java REST Assured

REST Assured (<https://rest-assured.io>) — это библиотека автоматизации тестирования REST API на Java. Она предлагает предметно-ориентированный язык (Domain-Specific Language, DSL) с синтаксисом Gherkin (Given, When, Then — «Дано, когда, тогда») для создания читаемых и удобных в обслуживании тестов API, а также использует средства проверки утверждений. REST Assured может работать с любым фреймворком тестирования, например, JUnit или TestNG.

Предположим, что в нашем гипотетическом сервисе управления заказами есть GET API `/items`, который возвращает список товаров и информацию о них:

GET: <https://eCommerce.com/items>

Response:

Status Code: 200

```
[
  {
    "SKU": "984058981",
    "Color": "Green",
    "Size": "M"
  }
]
```

Соответствующий код REST Assured DSL для вызова GET API и проверки кода состояния будет выглядеть так:

```
given().
    when().
        get("https://eCommerce.com/items").
    then().
        assertThat().statusCode(200);
```

Что может быть проще? Точно так же у вас есть DSL для тестирования POST, PUT и других HTTP-методов, поддерживаемых API.

Теперь настроим среду автоматизации тестирования API и напомним тест для проверки конечной точки GET `/items`. Эту конечную точку можно реализовать как заглушку, выполнив действия, описанные в главе 2.



Если вам нужны примеры API, чтобы попрактиковаться, то зайдите на сайт Any API (<https://anyapi.com>), где есть список, включающий 1400 общедоступных REST API.

Настройка и рабочий процесс. Как мы видели, когда настраивали среду автоматизации тестирования пользовательского интерфейса, тремя основными компонентами такой среды являются менеджер зависимостей (в нашем случае Maven), библиотека для выполнения требуемого типа тестирования (REST Assured) и фреймворк тестирования для создания и запуска тестов (задействуем TestNG). Создайте свою среду, объединив эти три компонента, как описано далее.

1. Создайте новый проект Maven, используя IntelliJ (или другую IDE по своему выбору). Подробности см. в пункте «Фреймворк Selenium WebDriver для Java» ранее в этой главе.
2. Добавьте зависимости TestNG и REST Assured в файл `pom.xml`. Найти необходимые параметры зависимостей можно в центральном репозитории Maven, как обсуждалось ранее.
3. Создайте новый пакет с именем `tests` в папке `/src/test/java` и новый тестовый класс с именем `ItemsTest`.
4. В примере 3.14 показан образец теста для проверки конечной точки `GET /items`.

Пример 3.14. Класс `ItemsTest` с тестом конечной точки `GET /items`

```
// ItemsTest.java

package apitests;

import org.testng.annotations.Test;

import static io.restassured.RestAssured.given;

public class ItemsTest {

    @Test
    public void verifyGetItemsEndpointReturnsSuccessStatusCode(){
        given().
            when().
            get("http://localhost:1000/items").
            then().
            assertThat().statusCode(200);
    }
}
```

Тест можно запустить из IDE или выполнив команду `mvn clean test` в терминале.

Как только базовая конфигурация заработает, вы сможете добавить тест для проверки конечной точки `POST /items`. Допустим, конечная точка `POST` принимает те же сведения об элементе в формате JSON и возвращает HTTP-ответ 201 при успешном добавлении элемента в новый заказ. Создайте заглушку на своем компьютере, выполнив действия, описанные в главе 2.

Чтобы передать JSON-код в теле запроса POST, создайте класс `dataObject` и сериализуйте его с помощью библиотеки сериализации JSON, например `jackson-databind`. Добавим его в свою среду.

1. Добавьте библиотеку `jackson-databind` в файл `pom.xml`.
2. Создайте новый пакет `dataObjects` в каталоге `/src/main/java` и добавьте новый класс `dataObject`, скажем `ItemDetails.java`. В примере 3.15 показан класс `ItemDetails`, представляющий тело JSON для запроса POST.

Пример 3.15. Класс `ItemDetails` как `dataObject`

```
// ItemDetails.java

package dataobjects;

import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonPropertyOrder;

@JsonPropertyOrder({"sku", "color", "size"})
public class ItemDetails {

    private String sku;
    private String color;
    private String size;

    public ItemDetails(String sku, String color, String size){
        this.sku = sku;
        this.color = color;
        this.size = size;
    }

    @JsonProperty("sku")
    public String getSku(){
        return sku;
    }

    @JsonProperty("color")
    public String getColor(){
        return color;
    }

    @JsonProperty("size")
    public String getSize(){
        return size;
    }
}
```

Обратите внимание на то, что библиотека `jackson-databind` позволяет предварительно определить ожидаемую структуру JSON-объекта с помощью аннотации `@JSONPropertyOrder`.

3. В тестовом классе можно использовать объект `ItemDetails` в качестве тела запроса POST. В примере 3.16 показан тест конечной точки POST `/items`.

Пример 3.16. Тест конечной точки POST `/items`

```
@Test
public void verifyPostItemsEndpointReturnsSuccessStatusCode(){

    ItemDetails greenShirt = new ItemDetails("98765490", "Green", "M");

    given().
        contentType(ContentType.JSON).
        body(greenShirt).
        log().body().
        when().
            post("http://localhost:1000/items").
        then().
            assertThat().
            statusCode(200);
}
```

После запуска теста метод `log().body()` запишет в логи тело запроса, чтобы вы могли проверить правильность сериализации. В этом примере мы подтвердили лишь `statusCode` ответа. Но вообще REST Assured позволяет отыскивать любые обязательные поля в теле ответа, как подробно описано в официальной документации (<https://oreil.ly/KIz1x>), и проверять их.

Модульные тесты

Модульные тесты тесно интегрированы с кодом приложения, поэтому фреймворк тестирования должен быть совместим с языком программирования, на котором написано приложение, например JUnit или TestNG для Java, NUnit для .NET, Jest или Mocha для JavaScript,RSpec для Ruby и т. д. Далее мы рассмотрим настройку JUnit. Предварительные условия для JUnit такие же, как и для тестов API.



Модульные тесты пишут только разработчики, но тестировщики тоже должны понимать их базовую структуру, чтобы осознанно планировать стратегию тестирования приложения. Цель данного упражнения — описание такого опыта для тестировщиков, поэтому этот подраздел будет простым.

JUnit

JUnit (<https://junit.org/junit5>) — очень популярный фреймворк модульного тестирования, созданный Кентом Бекем и Эрихом Гаммой в 1997 году. С тех пор он удовлетворяет весь спектр потребностей модульного тестирования и продолжает оставаться стандартным фреймворком модульного тестирования для Java де-факто.

JUnit предлагает такие возможности, как создание, организация и запуск тестов, а также генерирование отчетов. TestNG, еще один популярный фреймворк, был создан для устранения некоторых недостатков JUnit, но разработчики JUnit обновили и расширили его в последних выпусках, чтобы избавиться от них.

Вот некоторые из основных возможностей JUnit:

- аннотации тестов и жизненного цикла тестирования, такие как `@Test` для маркировки методов тестирования и `@BeforeEach`, `@BeforeAll`, `@AfterEach` и `@AfterAll` для маркировки действий по подготовке и освобождению ресурсов;
- аннотация `@DisplayName`, которая выводит читаемое имя каждого теста;
- нестандартные аннотации тегов, такие как `@Tag("smoke")`, которые при необходимости можно использовать для запуска только подмножества тестов;
- методы проверки утверждений, такие как `AssertTrue()`, `AssertEquals()`, `AssertAll()` и т. д.

Настройка и рабочий процесс. Напишем пару простых модульных тестов для сервиса поддержки клиентов в приложении электронной коммерции. Создайте новый проект Java и добавьте класс `CustomerManagement`, как показано в примере 3.17. В этом классе есть два метода: один добавляет нового клиента, а другой возвращает сведения о существующих клиентах. Далее мы добавим модульные тесты для этих двух методов.

Пример 3.17. Класс `CustomerManagement`

```
// CustomerManagement.java
```

```
package Customers;

import java.util.ArrayList;
import java.util.List;

public class CustomerManagement {

    private String firstName;
    private String lastName;
    private String age;

    private List<List<String>> customers = new ArrayList<List<String>>();

    public List<List<String>> getCustomers(){
        return customers;
    }

    // если передано пустое имя клиента,
    // то сгенерировать исключение, иначе добавить клиента
    public void addCustomers(List<String> customerDetails){
        if (customerDetails.get(0).isEmpty())
```



```

        throw new IllegalArgumentException();
        customers.add(customerDetails);
    }
}

```

Чтобы добавить модульные тесты, сделайте вот что.

1. Добавьте в файл `pom.xml` следующие зависимости:

```

<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.2</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>

```

2. Создайте файл `CustomerManagementTests.java` в папке `/src/main/test` и поместите в него новый тестовый класс.
3. Используйте аннотации и проверки утверждений JUnit для создания тестов, как показано в примере 3.18.

Пример 3.18. Файл `CustomerManagementTests.java` с тестами JUnit

```

package customersUnitTests;

import Customers.CustomerManagement;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;
import java.util.List;

@DisplayName("When managing new customers")
public class CustomerManagementTests {

    @Test
    @DisplayName("should return empty when there are no customers")
    public void shouldReturnEmptyWhenThereAreNoCustomers(){
        CustomerManagement customer = new CustomerManagement();
        List<List<String>> customers = customer.getCustomers();

        assertTrue(customers.isEmpty(), "Error: Customers exists");
    }
}

```

```
@Test
@DisplayName("should throw exception when customer name is invalid")
public void shouldThrowExceptionForInvalidInput(){
    List<String> newCustomer = new ArrayList<>();
    newCustomer.add("");
    newCustomer.add("Jackson");
    newCustomer.add("20");

    CustomerManagement customer = new CustomerManagement();
    IllegalArgumentException err =
        assertThrows(IllegalArgumentException.class, () ->
            customer.addCustomers(newCustomer));
}
```

Как можно видеть в примере 3.18, тег `@DisplayName` содержит читаемые описания тестов. Судя по этим описаниям, первый тест проверяет, возвращает ли метод `getCustomers()` пустое значение, когда нет ни одного существующего клиента, а второй — генерирует ли метод `addCustomers()` исключение `IllegalArgumentException` при попытке добавить клиента с пустым именем. Также обратите внимание на методы проверки исключений и возвращаемых значений.

Эти тесты можно запустить из IDE или из командной строки, используя команду `mvn clean test`. Результаты запуска тестов с их отображаемыми именами показаны на рис. 3.9.

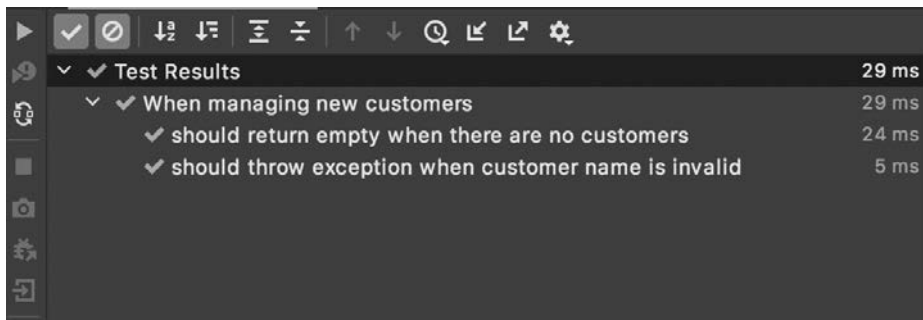


Рис. 3.9. В результатах выполнения тестов JUnit в среде разработки IntelliJ можно видеть их удобочитаемые наименования

Помимо JUnit, в зависимости от особенностей тестов вам могут потребоваться дополнительные возможности фреймворков разработки приложений (например, Spring Boot) и внешних библиотек, таких как Mockito для имитации вызовов сервисов, `jackson-databinder` для привязки данных, и т. д. Когда такие дополнительные возможности используются для доступа к внешней системе, например к базе данных, модульный тест превращается в интеграционный тест.

ХАРАКТЕРИСТИКИ ХОРОШИХ ТЕСТОВ

Перечисленные здесь характеристики справедливы для всех типов тестов, которые мы уже обсудили. Тесты, не обладающие этими характеристиками, легко превращаются в проблему при обслуживании.

- Тесты должны быть читабельными, иметь осмысленные названия методов и переменных, говорящие об их предназначении. Следуйте шаблону «Настрой, действуй, проверь» (Arrange, Act, Assert, AAA), согласно которому сначала нужно настроить тестовый сценарий в соответствии с предварительными условиями, затем выполнить предусмотренные им действия и, наконец, проверить ожидаемое поведение.
- Каждый тест должен проверять только один аспект поведения, чтобы не тратить много времени на выполнение и показывать корректный ожидаемый результат в случае неудачи.
- Тесты не должны зависеть друг от друга. Помните, что зависимые тесты склонны приводить к ошибкам зависимостей. Правильная подготовка и освобождение ресурсов каждого теста помогут сохранить их независимыми друг от друга и облегчат параллельное выполнение.
- Желательно, чтобы тесты не зависели от окружающей среды. Например, они не должны зависеть от статических данных в конкретной среде.
- Автоматизируйте процессы создания и запуска тестов, чтобы любой член команды мог получить код из репозитория и одной командой запустить тесты, не заботясь об управлении зависимостями вручную.

Дополнительные инструменты тестирования

В этом разделе мы рассмотрим еще несколько инструментов автоматизации тестирования: Pact — инструмент контрактного тестирования, Karate — инструмент BDD для создания сервисных тестов и некоторые инструменты автоматизации тестирования на основе ИИ и машинного обучения, которые сейчас приобретают все большую популярность. Это поможет вам получить более полное представление об инструментах автоматизации функционального тестирования и сделать правильный выбор, когда потребуется.

Pact

Pact (<https://docs.pact.io>) — популярный инструмент создания тестов на Java для проверки контрактов. Тесты также можно писать на Python, JavaScript, Go, Scala и других языках. Pact специально используется для *контрактного тестирования, ориентированного на потребителя* (consumer driven contract, CDC).

Потребитель — это приложение (например, сервис или веб-интерфейс), получающий информацию от другого приложения (например, сервиса или очереди сообщений). Приложение, предоставляющее информацию, называется *поставщиком*. Например, сервис управления заказами в приложении электронной коммерции получает сведения о товаре от сервиса управления информацией

о товарах, поэтому сервис управления заказами становится потребителем, а сервис управления информацией о товарах — поставщиком. Обратите внимание на то, что последний могут использовать многие другие потребители, не только сервис управления заказами. Кроме того, различным потребителям может требоваться разная информация. Например, сервис управления заказами может среди прочего запрашивать артикул каждого товара, но не запрашивать адрес его производителя, а другие потребители, напротив, могут запрашивать адрес производителя, но не запрашивать артикул товара.

Учитывая, что требования определяются потребителями, служба управления информацией о товарах может оказаться в ситуации, когда ей придется изменить свои контракты для удовлетворения нужд нового потребителя или нового требования, что создаст риск для службы управления заказами и других групп потребителей. Им нужен механизм, позволяющий проверять целостность контрактов службы управления информацией о товарах, особенно в части наборов возвращаемых атрибутов, чтобы в дальнейшем избежать проблем с интеграцией. Тестировщики или разработчики могут писать интеграционные или сервисные тесты, чтобы снизить риск, но такие тесты нередко оказываются нестабильными и медленными из-за зависимости от обоих приложений и к тому же сложными в настройке и обслуживании. Кстати, иногда поставщик и потребитель могут разрабатываться параллельно, а в таких случаях невозможно писать сквозные интеграционные или сервисные тесты. Контрактные тесты, ориентированные на потребителя, позволяют разрешить эти затруднения.

Как можно видеть на рис. 3.10, при тестировании контрактов, ориентированных на потребителя, каждая группа потребителей пишет тесты для контрактов, согласованных с поставщиком. В частности, тесты проверяют наличие атрибутов, ожидаемых этим потребителем, а не весь контракт. Затем эти тесты передаются команде поставщика, которая сравнивает их с реальными API поставщика и обеспечивает их соответствие ожиданиям потребителей. При обнаружении отклонений команда поставщика может по крайней мере предупредить соответствующего потребителя о необходимости изменить ожидания.

По сути, этот тип контрактного тестирования разбивает сквозное интеграционное тестирование на части следующим образом.

- Каждый потребитель пишет тесты на микро- и макроуровне для проверки своего функционального поведения, используя имитацию поставщика (см. рис. 3.10).
- Каждый потребитель пишет также контрактные тесты, применяющие имитацию поставщика, а поставщик запускает их.
- Поставщик пишет тесты на микро- и макроуровне для проверки своего функционального поведения.

Это упрощает некоторые проблемы, возникающие при написании сквозных интеграционных тестов и сервисных тестов, поскольку объем контрактных тестов меньше, и избавляет от зависимостей.

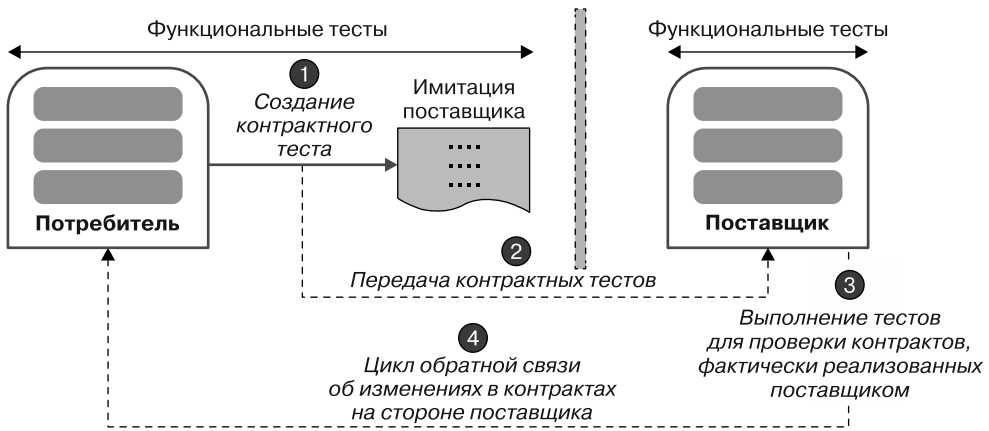


Рис. 3.10. Сценарий контрактного тестирования, ориентированного на потребителя

Pact позволяет полностью автоматизировать контрактное тестирование. Чтобы получить представление о рабочем процессе, используем тот же пример с сервисами управления заказами и информацией о товарах. Допустим, сервис управления заказами интегрируется с конечной точкой `GET /items` внешнего сервиса управления информацией о товарах для получения сведений о них, в частности складской номер, доступные размеры и цвета. Рабочий процесс Pact в обеих командах будет выглядеть так.

1. Команда сервиса управления заказами проверяет все интеграционные тесты. Например, некоторые из них будут предполагать, что конечная точка `/items` предоставляет ожидаемые сведения о товаре, если он существует, и пустой массив, если его не существует, и возвращает соответствующие коды ошибок (404, 500 и т. д.) для запросов, потерпевших неудачу.
2. Команда сервиса управления заказами создает имитации для этих тестов с помощью Pact.
3. Команда сервиса управления заказами пишет контрактные тесты, используя Pact для реализации имитаций, и проверяет в них конкретные атрибуты: коды состояния, складской номер, доступные размеры и цвета. Перед запуском этих тестов автоматически создается *pact-файл*. В нем фиксируются различные запросы к конечной точке `/items` и результаты проверки ожидаемых атрибутов в ответах.
4. Pact-файл автоматически передается команде сервиса управления информацией о товарах через ПО с открытым исходным кодом, называемое Pact Broker, которое следует настроить. Оно должно поддерживаться как командами потребителей, так и командой поставщиков. Команда разработки Pact также предлагает платную услугу под названием Pactflow, которая избавляет от необходимости настраивать и обслуживать Pact Broker. Чтобы упростить задачу, файлы можно передавать через общие папки.

5. Команда сервиса управления информацией о товарах пишет контрактный тест, чтобы получить pact-файл из Pact Broker, и настраивает тестовые данные в различных состояниях в соответствии с требованиями потребительских тестов. При запуске теста на стороне поставщика Pact, как описано в pact-файле, выполнит соответствующие запросы к фактическому сервису управления информацией о товарах и проверит фактические ответы.
6. Результаты тестирования на стороне поставщика передаются потребителю через Pact Broker, завершая полный цикл обратной связи без вмешательства человека.
7. Тесты Pact на стороне потребителя и поставщика интегрируются в конвейер CI, поэтому команды могут постоянно получать обратную связь.

В примере 3.19 показана возможная реализация потребительского теста Pact с помощью метода `pactMethod`. Прежде всего `pactMethod` устанавливает ожидаемое состояние конечной точки `/items`. Как можно заметить, это состояние, которое будет задействоваться тестом провайдера для инициации соответствующей настройки тестовых данных, описывается методом `given()`. Затем потребительский тест Pact вызывает имитацию конечной точки `/items`, как описано в `pactMethod`, и проверяет ответ со сведениями о товаре.

Пример 3.19. Пример потребительского теста с использованием Pact

```
@ExtendWith(PactConsumerTestExt.class)
public class ItemsPactConsumerTest {

    @Pact(consumer = "Order service", provider = "PIMService")
    RequestResponsePact getAvailableItemDetails(PactDslWithProvider builder) {
        return builder.given("items are available")
            .uponReceiving("get item details")
            .method("GET")
            .path("/items")
            .willRespondWith()
            .status(200)
            .headers(Map.of("Content-Type", "application/json; charset=utf-8"))
            .body(new JSONArrayMinLike(2, array -> {
                array.object(object -> {
                    object.stringType("SKU", "A091897654");
                    object.stringType("Color", "Green");
                    object.stringType("Size", "S");
                })
            })
            .build())
            .toPact();
    }

    @Test
    @PactTestFor(pactMethod = "getAvailableItemDetails")
    void getItemDetailsWhenItemsAreAvailable(MockServer mockServer) {
        // запускает имитацию конечной точки /items,
        // как определено методом ранее
    }
}
```

```

    RestTemplate restTemplate = new RestTemplateBuilder()
        .rootUri(mockServer.getUrl())
        .build();

    List<Item> items = new PIMService(restTemplate).getAvailableItemDetails();

    Item item1 = new Item("A091897654", "Green", "S");
    Item item2 = new Item("A091897654", "Green", "S");
    List<Item> expectedItems = List.of(item1, item2);
    assertEquals(expectedItems, items);
}

```

Этот тест создаст pact-файл, который будет передан поставщику через общую папку. Тест поставщика, показанный в примере 3.20, получает pact-файл, настраивает тестовые данные в методе с аннотацией `@State`, посылает запрос фактической конечной точке `/items`, как указано в pact-файле, и проверяет наличие в ответе тех же сведений и в том же формате, что и в примере 3.19.

Пример 3.20. Пример теста поставщика с использованием Pact

```

@Provider("PIMService")
@PactFolder("pacts")
@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)

public class ItemsPactProviderTest {

    @LocalServerPort
    int port;

    @MockBean
    private ItemRepository itemRepository;

    @BeforeEach
    void setUp(PactVerificationContext context) {
        context.setTarget(new HttpTestTarget("localhost", port));
    }

    @TestTemplate
    @ExtendWith(PactVerificationInvocationContextProvider.class)
    void verifyPact(PactVerificationContext context, HttpRequest request) {
        context.verifyInteraction();
    }

    @State("items are available")
    void setItemsAvailableState() {
        when(itemRepository.getItems()).thenReturn(
            List.of(new Item("A091897654", "Green", "S"),
                new Item("A091897654", "Green", "S")));
    }
}

```

Pact генерирует отчеты в формате HTML, которые можно интегрировать в конвейер CI. Pact-тесты обычно тесно привязаны к коду приложения, и для их создания и отладки может потребоваться знание используемых фреймворков разработки приложений, таких как Spring Boot.

Karate

Главная привлекательная черта Karate (<https://github.com/karatelabs/karate>) — уникальный способ создания сервисных тестов. Он предлагает применять predetermined операторы Gherkin (аналогичны операторам Cucumber), избавляющие от необходимости писать программный код. Этот инструмент не ограничивается тестированием API — он предназначен для поддержки сквозного автоматизированного тестирования пользовательского интерфейса, контрактов, настройки фиктивного сервера и т. д., но делает написание сервисных тестов гораздо более простым, чем вы можете себе представить. В примере 3.21 показан тот же тест, который мы задействовали для проверки конечной точки GET /items с помощью REST Assured, но написанный с использованием Karate.

Пример 3.21. Тест конечной точки GET /items, написанный с помощью Karate DSL

Feature: Order service should return item details

```
Scenario: verify GET items endpoint
  Given url 'http://localhost:1000/items'
  When method get
  Then status 200
```

Вот и все — три строки predetermined операторов Gherkin. Полный список операторов можно найти на странице Karate GitHub (<https://oreil.ly/K0zza>). Чтобы установить этот инструмент, достаточно импортировать архетип Maven во время создания проекта в IntelliJ.

Инструменты ИИ и машинного обучения в автоматизированном функциональном тестировании

В этой главе мы обсудили довольно много инструментов, и они способны удовлетворить самые широкие потребности в автоматизации функционального тестирования на всех уровнях приложения. Однако технологии искусственного интеллекта и машинного обучения привели к появлению новых инструментов, предлагающих дополнительную помощь в решении задач автоматизации тестирования, таких как разработка тестов, их обслуживание, анализ результатов тестирования и управление тестированием. В этом подразделе я представлю краткий обзор доступных на данный момент инструментов.

Создание тестов

Возможность задействовать искусственный интеллект и машинное обучение при написании тестов стала важной вехой в области тестирования, потому что дает возможность людям без навыков программирования создавать функциональные тесты пользовательского интерфейса. Test.ai, Functionize, Appvance, Testim и TestCraft — вот некоторые из коммерческих инструментов, предлагающих эту возможность.

Чтобы создавать тесты с их помощью, вам придется вручную пройти по пользовательскому сценарию на веб-сайте, а инструмент записи с поддержкой машинного обучения запомнит элементы и действия, выполняемые на каждом этапе, и создаст тесты в фоновом режиме. Преимущество инструмента на основе машинного обучения заключается в том, что он опознает элементы не только по их идентификаторам, но и по структурным и визуальным аспектам. Некоторые из этих инструментов могут помочь при обслуживании тестов и анализе первопричин ошибок, что значительно снижает нагрузку на систему автоматизации тестирования. Их также можно подключить к конвейеру CI, чтобы получать непрерывную обратную связь.

Обслуживание тестов

Вы когда-нибудь сталкивались с ситуацией, когда огромное количество функциональных тестов пользовательского интерфейса терпело неудачу из-за изменения идентификатора одного элемента? Чаще всего меняется только идентификатор элемента, а его функциональность и внешний вид остаются неизменными. И все же функциональные тесты пользовательского интерфейса потерпят неудачу, потому что они обычно полагаются на идентификатор элемента. У меня такое случилось, и не раз, и когда это происходило, я задавалась вопросом: существуют ли инструменты, способные автоматически исправить такие небольшие изменения и сэкономить мне время?

Теперь в инструментах автоматизации тестирования на базе искусственного интеллекта и машинного обучения, таких как Test.ai (<https://test.ai>) и Functionize (<https://www.functionize.com/test-maintenance>), доступна функция автокоррекции, называемая *самовосстановлением*.

Как упоминалось ранее, инструмент записи на основе машинного обучения фиксирует структурные и визуальные аспекты элементов пользовательского интерфейса вместе с их идентификаторами. Когда идентификатор элемента изменяется, он по-прежнему благополучно идентифицируется тестом, и инструменты просто запрашивают ваше одобрение на обновление значения идентификатора в тестовых сценариях.

Анализ отчетов с результатами тестирования

Как упоминалось ранее, мне доводилось видеть крупные корпоративные проекты, в которых применялись сотни автоматизированных тестов на основе пользовательского интерфейса, выполняющиеся всю ночь, а специальная группа тестировщиков анализировала результаты утром. Команда могла часами выяснять коренные причины неудач тестирования. Чаще всего их было три: ошибки в коде, изменения в новых функциях или проблемы среды. Выяснив причины, команда составляла отчеты об ошибках, исправляла тестовые сценарии с учетом изменений в новых функциях и связывалась с командой поддержки инфраструктуры, чтобы решить проблемы среды. Они решали одни и те же задачи день за днем. Им очень пригодился бы ReportPortal (<https://oreil.ly/frHa1>) — инструмент с открытым исходным кодом для анализа отчетов с результатами тестирования!

У ReportPortal есть функция автоматического анализа на основе машинного обучения, которая читает логи неудачных тестов и распределяет их по таким категориям, как ошибки, проблемы в тестовых сценариях и проблемы среды. Алгоритм машинного обучения изучает данные из логов ранее проанализированных неудачных тестов. Конечно, для этого требуется заранее вручную проверить и маркировать предыдущие неудачи тестирования. Но как только такие данные с анализом сбоев тестов будут готовы, автоматический анализатор обучится на них и начнет точно выявлять сбои тестов, экономя команде время.

Управление тестированием

Надлежащий охват прикладного кода тестированием на разных уровнях оказывается актуальной проблемой для всех команд. Специалисты могут радоваться огромному проценту охвата функциональными или модульными тестами, но не знать, что какой-то модуль вообще не тестируется. Управление тестированием заключается в размещении нужных тестов на нужных уровнях и внедрении контрольных показателей качества на каждом из них. Для этого требуются данные со всех уровней, включая перечень функций, не охваченных тестированием. Именно этой цели служит SeaLights (<https://oreil.ly/d9WtY>) — инструмент управления тестированием на базе искусственного интеллекта и машинного обучения: он показывает показатели охвата тестами на всех уровнях, определяет области кода с плохим охватом, выявляет риски для качества путем сопоставления данных выполнения тестов и охвата тестами, а также предоставляет множество других функций, связанных с управлением качеством.

Таковы возможности, обеспечиваемые технологиями искусственного интеллекта и машинного обучения в области автоматизации тестирования. Подводя итог, можно сказать, что их помощь становится все более существенной, а они сами продолжают развиваться. Там, где это возможно, желательно использовать такие

инструменты для избавления членов команды от рутинных задач, чтобы они могли сосредоточиться на задачах более высокого порядка, таких как планирование, инновации, безопасность, производительность и т. д.

Перспективы

Мы довольно далеко проникли в область автоматизации функционального тестирования, но прежде, чем завершить главу, я хотела бы обратить ваше внимание еще на несколько ключевых тем: антипаттерны в автоматизированном функциональном тестировании, охват автоматизированными тестами и, в частности, что значит иметь стопроцентный охват автоматизированными тестами.

Антипаттерны, которых нужно сторониться

Даже потратив уйму времени и сил на разработку правильной стратегии автоматизированного функционального тестирования и внедрение фреймворков тестирования на надлежащих уровнях, вы должны понимать, что ваш путь в автоматизации функционального тестирования только начался. На протяжении всего срока поставки вы должны следить за появлением антипаттернов по мере того, как команда разрабатывает все больше и больше тестов. Как показывает мой опыт, стать жертвой этих антипаттернов легко, поэтому крайне важно следить за ранними симптомами их появления. В этом разделе мы обсудим распространенные антипаттерны «рожок мороженого» и «слоеный пирог» (рис. 3.11), а также их симптомы и советы по их преодолению.

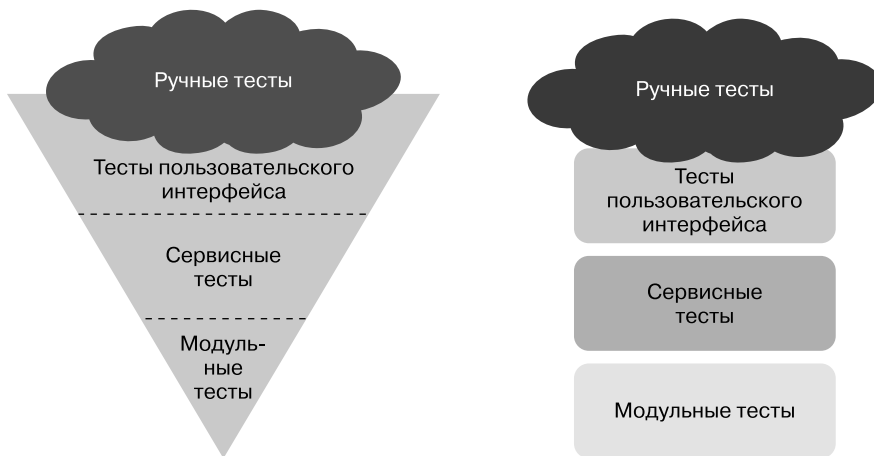


Рис. 3.11. Антипаттерны в автоматизированном функциональном тестировании

«Рожок мороженого»

Если перевернуть тестовую пирамиду, она будет выглядеть как конус. Этот антипаттерн, в котором больше тестов на макроуровне, управляемых пользовательским интерфейсом, и очень мало тестов на микроуровне, называется «рожок мороженого» (<https://oreil.ly/zoesB>). Вы можете заметить его по следующим симптомам:

- длительное ожидание информации с результатами тестирования;
- выявление дефектов на поздних этапах цикла, иногда только на этапе тестирования релиза;
- тщательное ручное тестирование, необходимое для получения обратной связи, несмотря на наличие автоматизированных тестов;
- разочарование в команде по поводу автоматизированных тестов из-за того, что автоматизация тестов пользовательского интерфейса не принесла нужных результатов.



Самый ранний признак, обнаружив который можно предотвратить скатывание вашей команды к данному антипаттерну, — это обнаружение дефектов во время ручного регрессионного тестирования историй. Немедленно проанализируйте первопричины их появления и как можно скорее исправьте методы работы вашей команды.

«Слоеный пирог»

Когда тесты дублируются на нескольких уровнях, вместо пирамиды получается своеобразный слоеный пирог (<https://oreil.ly/tzJzw>), имеющий широкий нижний слой, не менее широкую середину и еще более широкий верхний слой. Такая дезорганизация обычно возникает, когда команды разработчиков и тестировщиков действуют несогласованно. Например, разработчики добавляют модульные тесты для проверки всех ошибочных входных данных, а тестировщики — точно такие же тесты на уровень пользовательского интерфейса.

Узнать этот антипаттерн можно по большим затратам времени, необходимым для выпуска даже крошечной новой функции. Также можно заметить игры с перекаладыванием ответственности, когда кто-то один ожидает, что другой добавит соответствующие тесты при обнаружении ошибки.



Простой способ избежать этого антипаттерна — провести короткое обсуждение в команде и определить, какие тесты должны создаваться на каждом уровне. Для этого прекрасно подходит начальное собрание для обсуждения пользовательских историй, после которого следует задокументировать результаты обсуждения в карточках пользовательских историй.

Стопроцентный охват кода автоматизированными тестами

Обычно команды следят за долей прикладного кода, охваченного автоматизированными тестами, и считают высокий процент подтверждением применения хороших практик разработки ПО. Процент охвата автоматизированными тестами рассчитывается путем сбора всех тестовых сценариев, их маркировки как автоматизированных или нет и выполнения простых арифметических вычислений для получения процентного значения. Команды часто ставят перед собой цель достичь стопроцентного охвата, но при этом упускают из виду несколько важных моментов.

ОХВАТ КОДА И МУТАЦИОННОЕ ТЕСТИРОВАНИЕ

Традиционная мера охвата кода отличается от меры охвата автоматизированными тестами. Охват кода сообщает, есть ли строки кода, которые не будут выполняться существующими модульными тестами. Другими словами, она идентифицирует строки кода, не охваченные тестами. Инструменты измерения охвата кода, такие как JaCoCo и Cobertura, можно интегрировать в конвейер сборки CI и вызывать сбой, когда процент охвата оказывается ниже определенного порога, чтобы предотвратить просачивание непроверенного кода дальше этапа сборки. Однако высокая доля охвата кода не обязательно означает, что все тесты автоматизированы.

Для выявления пропущенных тестовых сценариев в модульном тестировании используется метод, называемый *мутационным тестированием*. Оно изменяет код приложения и проверяет, успешно ли выполняются тесты. Например, из исходного кода удаляются вызовы ничего не возвращающих методов и модульные тесты запускаются снова. Если тестирование завершилось с ошибкой, то говорят, что мутация убита, в противном случае — что выжила. PIT (<https://oreil.ly/aeGI0>) — популярный инструмент мутационного тестирования, который можно добавить как зависимость Maven и запускать из командной строки. В своем отчете он сообщит, какие тестовые сценарии выжили, а также даст общую оценку мутаций приложения. Мутационное тестирование, хотя и очень эффективное, требует много времени, поэтому пользоваться им нужно с умом.

Первое, что я хочу отметить относительно процента охвата кода автоматизированными тестами: даже стопроцентный охват не гарантирует отсутствия ошибок в приложении! Процент — это просто мера, показывающая, сколько известных тестовых сценариев автоматизировано. Позже вы можете обнаружить неизвестные на данный момент случаи. Важно указать на это заинтересованным сторонам бизнеса и членам команды, потому что иначе, обнаружив позднее критическую ошибку, они могут усомниться в надежности набора автоматизированных тестов и необходимости тратить на них свои силы и время. Также важно дать им понять, что цели наблюдения за этой мерой — выявление отставания в автоматизации (в идеале его не будет) и планирование выполнения этих задач в предстоящих итерациях. Эту меру можно использовать и для того, чтобы вовремя заметить скатывание к одному из антипаттернов, упомянутых в предыдущем разделе.

Второй момент: отслеживая степень автоматизации, следите за тем, все ли области приложения охвачены автоматизированным тестированием. Это особенно важно

при разработке крупномасштабных приложений, когда над разными компонентами работают разные команды. Общий процент охвата может быть высоким (скажем, больше 80 %), но при этом одни модули могут быть вообще не охвачены тестированием, а другие — иметь охват, близкий к 100 %.

Третий важный момент: при расчете этой меры включайте как функциональные, так и межфункциональные тестовые сценарии. Чаще всего межфункциональные тесты не влияют на процент охвата, что позже приводит к ошибкам (подробнее об автоматизации межфункциональных тестов вы узнаете в следующих главах).

И наконец, несмотря на стремление автоматизировать все тестовые сценарии, в некоторых случаях — в зависимости от характера приложения, среды, величины затрат на автоматизацию и т. д., — может оказаться невозможным достичь стопроцентного охвата кода автоматизированными тестами. В таких случаях следует с особым вниманием отнестись к неавтоматизированным тестовым примерам и добавить их в список ручного тестирования. Но такой список не должен быть слишком длинным — нежелательно тратить по 1200 минут на операционное тестирование, о чем я предупреждала в начале главы!

Преимущества тщательного отслеживания и обеспечения надлежащего охвата автоматизацией начнут проявляться по мере роста проекта, особенно если он растянется на несколько лет. Как говорится, люди приходят и уходят, а код остается, и часто автоматизированные тесты оказываются единственной заслуживающей доверия живой документацией о функциях приложения. Поэтому ваши усилия по созданию автоматизированных тестов будут ценной инвестицией не только для проекта, но и для вас и ваших будущих товарищей по команде.

Ключевые выводы

- Автоматизированное тестирование — это практика использования инструментов для проверки ожидаемого поведения приложения и получения быстрой обратной связи во время разработки.
- Разумный способ сбалансировать объем тестирования в проекте — выполнить ручное исследовательское тестирование, отыскать тестовые сценарии и автоматизировать их, чтобы впоследствии использовать для выполнения регресса.
- Область применения автоматизированного функционального тестирования намного шире сферы применения общепринятого функционального тестирования на основе пользовательского интерфейса. Модульные, интеграционные, контрактные, функциональные тесты, тесты пользовательского интерфейса и сквозные тесты — все это разные виды тестов на микро- и макроуровне, которые при правильном сочетании обеспечивают быструю обратную связь.

- Пирамида тестирования — идеальная цель, к которой следует стремиться при разработке стратегии автоматизированного функционального тестирования. Широкое основание с тестами микроуровня и постепенное сокращение количества тестов макроуровня по мере расширения охватываемой ими области — лучший способ сократить время, необходимое для создания и выполнения тестов.
- Для упрощения разработки, обслуживания и анализа результатов автоматизированных функциональных тестов было разработано несколько инструментов, в том числе инструменты на основе искусственного интеллекта и машинного обучения.
- Вы можете затратить массу времени и сил на создание базы для автоматизации тестирования на разных уровнях, но работа на этом не заканчивается. Вы должны постоянно следить за симптомами появления антипаттернов, таких как «рожок мороженого» и «слоеный пирог».
- Очень важно отслеживать долю прикладного кода, охваченного автоматизацией, чтобы усилия по автоматизации не пропали даром в ажиотаже разработки. Но при этом имейте в виду, что высокий процент охвата кода автоматизированными тестами может создать у команды ложное ощущение безопасности. Поэтому важно не ограничиваться цифрами и постараться обеспечить равномерный охват всех областей приложения.

ГЛАВА 4

Непрерывное тестирование

От быстрой обратной связи мало проку, если она не непрерывна!

В предыдущей главе мы обсудили ускорение цикла обратной связи добавлением тестов на разных уровнях приложения. Но такую быструю обратную связь крайне важно получать не порциями от случая к случаю, а *постоянно*, чтобы иметь возможность регулировать качество приложения на протяжении всего цикла разработки. Эта глава посвящена разработке практики непрерывного тестирования.

Непрерывное тестирование (continuous testing, СТ) — это процесс проверки качества приложения с использованием ручных и автоматизированных методов тестирования после добавления каждого изменения и оповещения команды, когда оно приводит к ухудшению качества. Например, когда производительность каких-то функций отклоняется от ожидаемых показателей, процесс СТ немедленно уведомит об этом сообщениями о неудачах тестов производительности. Благодаря этому команда сможет быстро исправить проблемы, пока они еще относительно малы и управляемы. В отсутствие подобного непрерывного цикла обратной связи проблемы могут долго оставаться незамеченными и просачиваться на более глубокие уровни кода, что приведет к росту затрат сил и времени, необходимых на их устранение.

Процесс СТ в значительной степени опирается на практику *непрерывной интеграции* (continuous integration, CI) и автоматизированного тестирования каждого изменения. Внедрение CI вместе с СТ позволяет команде осуществлять *непрерывную доставку* (continuous delivery, CD). В конечном итоге трио CI, CD и СТ делает команду высокоэффективной по четырем ключевым показателям: *время выполнения заказа, частота развертывания, среднее время восстановления и процент неудачных изменений*. Эти показатели, которые мы рассмотрим ближе к концу главы, дают представление о качестве работы команды по доставке.

Эта глава поможет вам приобрести навыки, необходимые для организации процесса СТ в команде. Вы узнаете о процессах и стратегиях CI/CD/СТ для создания

нескольких циклов обратной связи по различным характеристикам качества. Здесь также приводится пошаговое упражнение по настройке сервера CI и интеграции автоматических тестов.

Введение

Этот раздел знакомит с терминологией и общим процессом CI/CD/CT, а также представляет фундаментальные принципы и правила, которые следует выполнять и соблюдать, чтобы процесс был успешным. Начнем со знакомства с CI.

Введение в непрерывную интеграцию

Мартин Фаулер (<https://oreil.ly/Z2kjh>), автор полудюжины книг, включая *Refactoring: Improving the Design of Existing Code*¹ (Addison Wesley), и главный научный сотрудник Thoughtworks, описывает непрерывную интеграцию как «практику разработки программного обеспечения, согласно которой члены команды часто интегрируют свою работу, обычно каждый день, что приводит к выполнению нескольких интеграций ежедневно». Рассмотрим пример, иллюстрирующий преимущества такой практики.

Двое коллег по команде, Элли и Боб, независимо друг от друга начали разработку начальной страницы и страницы входа. Работа началась утром, и к полудню Элли завершила реализацию базового процесса входа, а Боб закончил базовую структуру домашней страницы. Они оба протестировали соответствующие функции на своих локальных машинах и продолжили работу. К концу дня Элли завершила реализацию своей функции, которая после успешного входа переадресовывала пользователя на пустую начальную страницу, потому что к этому моменту та еще не была готова. Аналогично Боб закончил реализацию начальной страницы, жестко закодировав имя пользователя в приветственном сообщении, потому что информация о пользователе, выполнившем вход, была ему недоступна.

На следующий день они оба сообщили, что их функции готовы! Но так ли это на самом деле? Кто из двух разработчиков должен отвечать за интеграцию страниц? Должны ли они создать отдельную пользовательскую историю для каждого сценария интеграции в приложении? Если да, то готовы ли они к дублированию затрат сил и времени на тестирование истории интеграции? Или им следует отложить тестирование до завершения интеграции? Именно такие вопросы неявно решаются с помощью непрерывной интеграции.

Если следовать CI, то Элли и Боб должны делиться результатами своей работы в течение дня (в конце концов, к полудню у обоих уже был готов базовый скелет

¹ Фаулер М. Рефакторинг. Улучшение проекта существующего кода. — М., 2019.

их функций). Боб сможет добавить необходимый код для абстрагирования имени пользователя после входа в систему (например, из токена JSON или JWT), а Элли — переадресовать пользователя на фактическую начальную страницу после успешного входа. Тогда приложение действительно будет пригодным для применения и тестирования!

В этом конкретном примере интеграция двух страниц на следующий день потребует совсем небольших дополнительных затрат. Однако, когда код накапливается и интегрируется на более поздних этапах цикла разработки, интеграционное тестирование усложняется и отнимает много времени. Более того, чем позднее выполняется тестирование, тем выше вероятность обнаружить запутанные проблемы, которые трудно, а иногда и невозможно исправить без переписывания значительной части программного кода. Впоследствии это воспитает у членов команды страх перед интеграцией — часто неперенный спутник отсроченной интеграции!

Практика непрерывной интеграции, по сути, призвана снизить такие риски и избавить команду от переписываний и исправлений. Это не избавляет полностью от дефектов интеграции, но облегчает их обнаружение и исправление на ранней стадии, когда они только зарождаются.

Процесс CI/CT/CD

Начнем со знакомства с процессами непрерывной интеграции и тестирования. Позже мы увидим, как они объединяются для формирования процесса непрерывной доставки.

Процесс CI/CT основан на четырех отдельных компонентах:

- на системе управления версиями (Version Control System, VCS), которая хранит весь код приложения и служит центральным хранилищем, откуда все члены команды получают последнюю версию кода и где они могут непрерывно интегрировать результаты своей работы;
- автоматизированных функциональных и межфункциональных тестах, проверяющих приложение;
- сервере CI, который автоматически тестирует последнюю версию кода приложения после сохранения каждого изменения;
- инфраструктуре, в которой размещены сервер CI и приложение.

Рабочий процесс непрерывной интеграции и тестирования начинается с разработчика, который, закончив реализацию некоторой части функциональности, сохраняет изменения в общую систему управления версиями, например Git либо SVN. Получив изменения, VCS направляет их в процесс непрерывного тестирования, который компилирует код приложения и применяет к нему автоматизированные тесты с помощью сервера CI, например Jenkins или GoCD. Если все тесты выпол-

нились успешно, то новые изменения считаются полностью интегрированными. При возникновении сбоев владелец соответствующего кода устраняет проблемы как можно быстрее. Иногда VCS несколько раз возвращает изменения на доработку, пока не будут ликвидированы все обнаруженные проблемы. Это делается главным образом для того, чтобы другие не могли извлечь код с проблемами и интегрировать свою работу поверх него.

ПРЕИМУЩЕСТВА СИСТЕМ УПРАВЛЕНИЯ ВЕРСИЯМИ

Задумывались ли вы о том, как команды разрабатывали приложения до появления VCS? Некоторые использовали общие диски, другие напрямую пересылали свой код на центральный сервер, где размещалась вся база кода! Для устранения всех этих неудобств в 1960-х была создана первая система управления версиями, получившая название «система управления исходным кодом» (Source Code Control System, SCCS). С тех пор возможности систем управления версиями заметно расширились, появились новые функции, помогающие устранить множество проблем и предлагающие огромные преимущества для интеграции работы.

Вот некоторые из наиболее существенных преимуществ.

- Система управления версиями отслеживает каждое изменение, будь то добавление, удаление или модификация кода, в отдельной базе данных. Она хранит всю историю изменений и, следовательно, значительно облегчает анализ первопричин проблем.
- Сохраняя изменения отдельно, VCS позволяет командам вернуться к ранее работавшей версии приложения в случае возникновения проблем.
- Изменения в VCS могут привязываться к пользовательской истории или карте дефектов. Благодаря этому команда может отслеживать изменения в пользовательской истории и понимать контекст написанного кода и эволюцию функции с течением времени.
- Иногда членам команд приходится работать над общей областью кода. Для этой цели VCS позволяет создавать ответвления (<https://oreil.ly/Ma8Ft>) от основной базы кода, дополнять их и объединять с основной базой кода позже. Однако долгоживущие ответвления считаются антипаттерном.

Как показано на рис. 4.1, Элли до полудня помещает свой код с реализацией функции входа вместе с тестами в общую систему управления версий в рамках коммита C_n .



Коммит в Git VCS — это сохранение состояния всей базы кода в определенный момент. При использовании непрерывной интеграции рекомендуется сохранять изменения небольшими порциями в виде независимых коммитов на локальном компьютере. Когда функциональность достигает некоторого законченного логического состояния, например завершения реализации базовой функции входа, коммиты должны быть отправлены в общий репозиторий VCS. Только после передачи изменений в VCS начинаются процессы CI и тестирования.

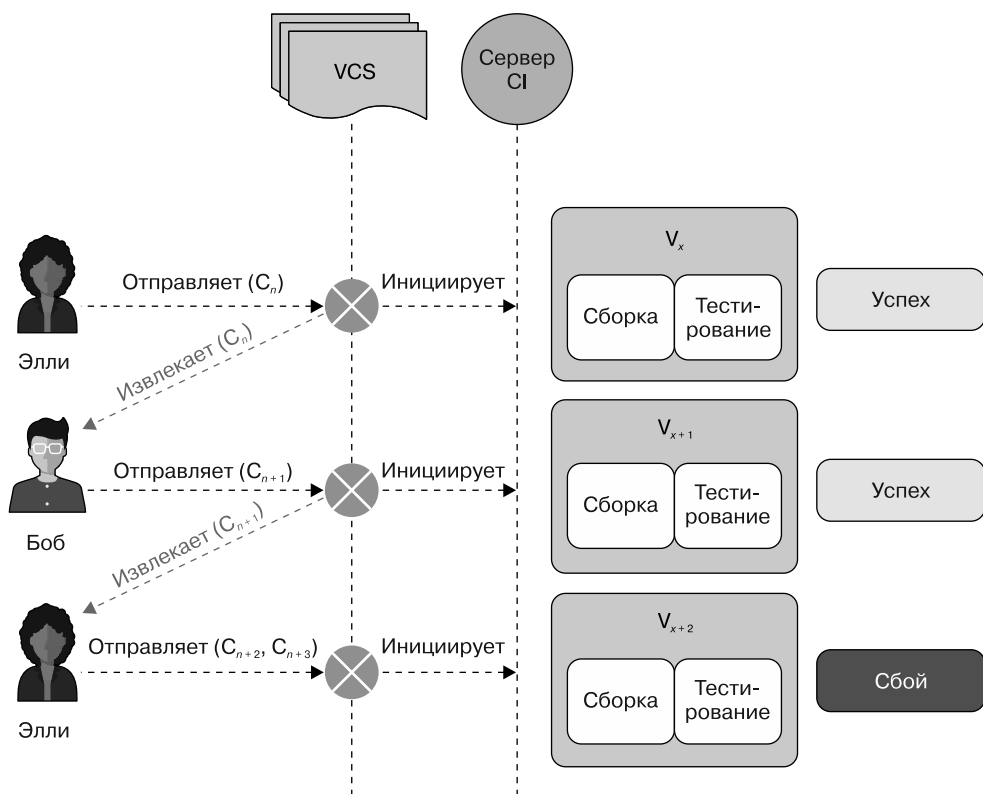


Рис. 4.1. Компоненты процесса непрерывной интеграции и тестирования

Новое изменение C_n инициирует запуск отдельного конвейера на сервере CI. Каждый конвейер состоит из множества последовательных этапов. Первый — этап *сборки и тестирования*, на котором компилируется приложение и выполняются автоматизированные тесты. К ним относятся все тесты микро- и макроуровня, обсуждавшиеся в главе 3, а также тесты, проверяющие аспекты качества приложения (производительность, безопасность и т. д.), которые мы обсудим в следующих главах. После завершения этого этапа результаты теста передаются Элли. В этом случае код Элли был успешно интегрирован, и она продолжает работать над функцией входа.

Позднее в тот же день, после получения последних изменений (C_n) из общей VCS, Боб отправляет коммит C_{n+1} с изменениями начальной страницы. Таким образом, C_{n+1} — это сохранение состояния базы кода приложения, включающего новые изменения, сделанные Элли и Бобом. Этот коммит запускает этап сборки и тестирования в процессе CI. Тесты, проверяющие изменение C_{n+1} , гарантируют, что новые изменения от Боба не нарушили ни одну из предыдущих функций,

включая последний коммит Элли, поскольку она также добавила тесты для проверки функции входа. К счастью, все тесты выполнены успешно. Однако (см. рис. 4.1) изменения, которые сделала Элли, C_{n+2} и C_{n+3} , нарушили интеграцию, и тесты потерпели неудачу. Теперь она должна исправить проблему, прежде чем продолжить работу, потому что внесла ошибку в общую систему управления версиями. Элли может отправить исправление в виде еще одного коммита, и процесс продолжится.

Представьте тот же рабочий процесс в большой распределенной команде, и вы поймете, насколько непрерывная интеграция упрощает возможность совместного использования и беспрепятственной интеграции плодов труда всех членов команды. Кроме того, в крупномасштабных приложениях обычно имеется несколько взаимозависимых компонентов, требующих полноценного интеграционного тестирования, и процесс непрерывного тестирования обеспечит столь необходимую уверенность в качестве их интеграции!

Благодаря уверенности, которую дает автоматизированный процесс интеграции и тестирования, команда оказывается в привилегированном положении, позволяющем выпускать код в промышленную эксплуатацию в любой момент, когда этого потребует бизнес. Другими словами, команда готова к непрерывной доставке.

Непрерывная доставка зависит от строгого соблюдения процессов непрерывной интеграции и тестирования, гарантирующих постоянную готовность приложения к выпуску. Но, помимо них, необходим также механизм автоматизированного развертывания, который можно запустить одним щелчком кнопкой мыши, чтобы развернуть приложение в выбранной среде, будь то среда отдела контроля качества или промышленная среда. На рис. 4.2 показан процесс непрерывной доставки.

Как видите, процесс непрерывной доставки включает процессы CI/CT, а также конвейеры развертывания. Эти конвейеры также являются этапами, настраиваемыми на сервере CI. Они выполняют развертывание выбранной версии артефактов приложения в требуемой среде.

Сервер CI перечисляет все коммиты и сообщает результаты их тестирования. Только если все тесты для коммита (или набора коммитов) выполняются успешно, он даст возможность развернуть эту конкретную версию приложения (V). Предположим, что команда Элли хочет получить отзывы от компании о функциях входа, добавленных как часть коммита C_n . Они могут нажать кнопку «Deploy V_x », как показано на рис. 4.2, и выбрать среду приемочного тестирования на стороне пользователя (user acceptance testing, UAT). В этой среде будут развернуты только изменения, добавленные к указанному моменту, то есть изменения C_{n+1} . Боба и более поздние не будут развернуты. Изменения C_{n+2} и C_{n+3} вообще не будут доступны для развертывания, потому что их тестирование завершилось неудачей.

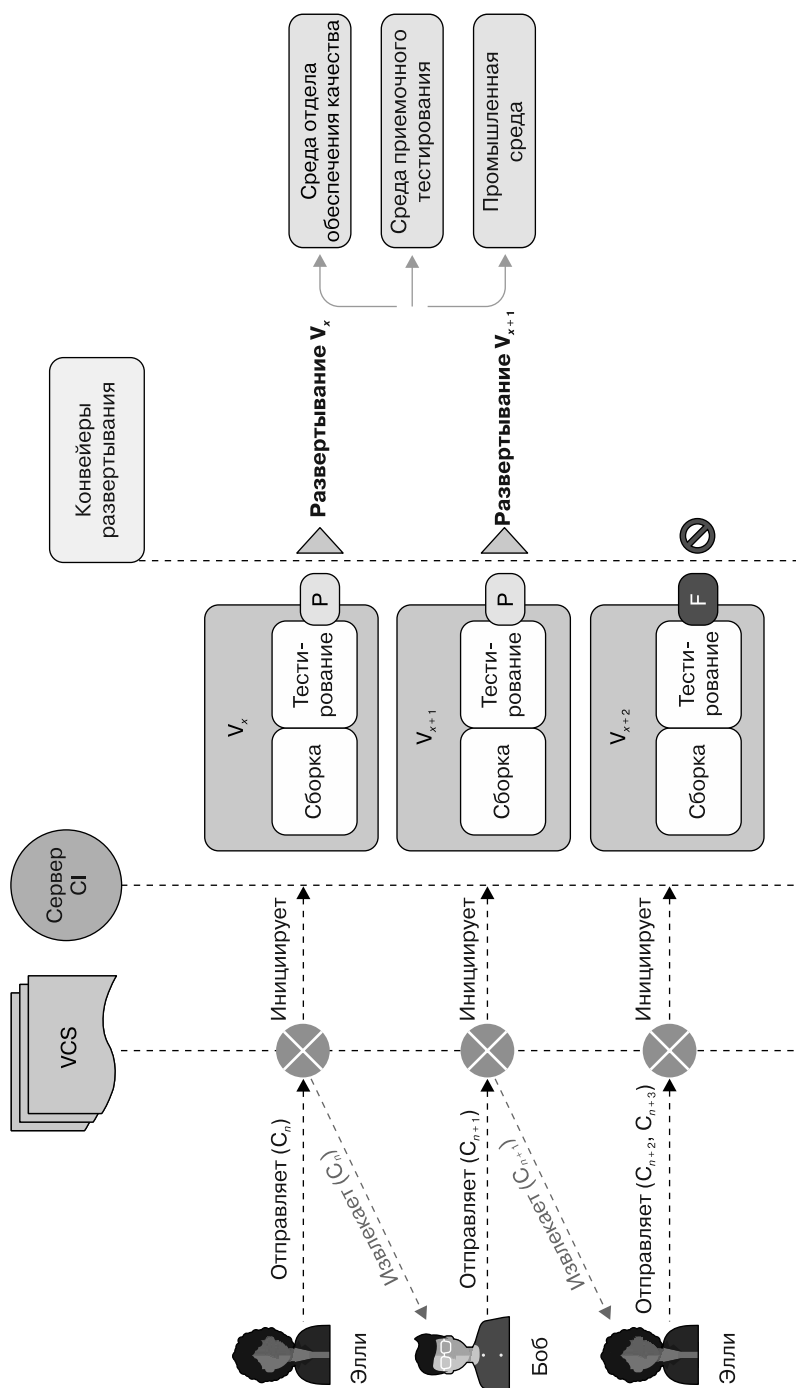


Рис. 4.2. Непрерывная доставка с конвейерами CI, СТ и развертывания

Такая настройка непрерывной доставки решает многие проблемы, но одно из наиболее важных преимуществ — возможность выводить на рынок функции продукта в нужное время. Часто задержки с выпуском новых функций приводят к потере доходов и уходу клиентов к конкурентам. Кроме того, для команды процесс развертывания становится полностью автоматизированным, что снижает зависимость от определенных людей, которые творят чудеса в день развертывания, — любой желающий может легко и без проблем выполнить развертывание в любой среде в любое время. Также автоматизация развертывания снижает риск несовместимости библиотек, отсутствия конфигурации или ошибок в ней и недостаточности документации.

НЕПРЕРЫВНОЕ РАЗВЕРТЫВАНИЕ И НЕПРЕРЫВНАЯ ДОСТАВКА

Непрерывное развертывание отличается от непрерывной доставки. Непрерывное развертывание предполагает наличие автоматизированных конвейеров развертывания, которые автоматически отправляют каждый коммит в эксплуатацию после непрерывного тестирования. Другими словами, функция, которую вы только что отправили в репозиторий, сразу же станет доступна реальным конечным пользователям на производстве. Непрерывная доставка, в свою очередь, означает лишь готовность в любой момент развернуть приложение в промышленной среде. Непрерывная доставка подходит в случаях, когда в компании установлены определенные даты выпуска функций. Иногда компании даже делают публичные объявления о вводе новой функции.

Принципы и правила

Теперь, обсудив процессы CI/CD/CT, следует отметить, что они могут принести плоды, только если все члены команды будут следовать набору четко определенных принципов и правил. В конце концов, таков автоматизированный способ совместной работы, будь то автоматизированные тесты, прикладной код или конфигурации инфраструктур. Команда должна установить набор принципов в начале цикла реализации и продолжать укреплять их на протяжении всего процесса. Вот минимальный набор принципов и правил, которые команда должна соблюдать, чтобы добиться успеха.

Часто фиксировать код

Члены команды должны фиксировать код в системе управления версиями по завершении реализации каждой небольшой части функциональности, чтобы она была протестирована и предоставлена другим для продолжения разработки на ее основе.

Всегда фиксируйте код вместе с автоматизированными тестами для его проверки

При передаче в репозиторий всякий новый фрагмент кода должен сопровождаться автоматическими тестами. Мартин Фаулер называет эту практику *самопроверяемым кодом* (<https://oreil.ly/9QNlb>). Например, как мы видели ранее, Элли зафиксировала свою функцию входа вместе с соответствующими тестами. Это дало возможность убедиться, что код в ее коммите сохранил работоспособность, когда позднее Боб зафиксировал свой код.

Придерживайтесь практики непрерывного интеграционного тестирования

Каждый член команды должен убедиться, что его решение благополучно прошло процесс непрерывного тестирования, прежде чем переходить к следующему набору задач. Если тесты потерпят неудачу, то возникшие проблемы нужно немедленно исправить. Согласно практике непрерывного интеграционного тестирования, предложенной Мартином Фаулером (<https://oreil.ly/lA0uR>), дефекты в сборке должны быть исправлены в течение 10 мин. Если это невозможно, то коммит с неработоспособным кодом следует отменить, чтобы оставить код работоспособным (или, как говорят тестировщики, зеленым).

Не игнорируйте и не отключайте неудачные тесты

Чтобы успешно пройти этапы сборки и тестирования, члены команды не должны отключать и игнорировать неудачные тесты. Несмотря на очевидность причин, почему этого не следует делать, такое происходит не так уж редко.

Не сохраняйте в репозитории неработающую сборку

Команда не должна отправлять свой код в репозиторий, если на этапе сборки или тестирования обнаружились проблемы. Продолжение работы с уже неработоспособным кодом приведет к повторному сбою тестов. Это только обременит команду дополнительной задачей по поиску изменений, приведших к сбою.

Возьмите на себя ответственность за все неудачи

Когда тесты терпят неудачу при проверке кода, который никто не изменял, но ошибка появилась после внесения изменений в какой-то другой код, то ответственность за исправление сборки должна ложиться на авторов этих изменений. При необходимости они могут объединиться еще с кем-то, обладающим необходимыми знаниями, чтобы исправить ошибку, но в любом случае решить проблему перед переходом к следующей задаче — обязательное условие. Это важно, потому что часто ответственность за исправление неудачных тестов перекладывается на других, что приводит к задержке решения проблем. Иногда тесты не выполняются в конвейере CI по несколько дней из-за того, что

проблема не была устранена. Из-за этого процесс непрерывного тестирования дает неполную или даже ложную информацию об изменениях, внесенных в этот период.

Ради собственной выгоды многие команды применяют еще более строгие методы. Например, требуют, чтобы все тесты микро- и макроуровня выполнялись на локальных машинах перед отправкой коммита в VCS, чтобы этап сборки и тестирования не запускался, если коммит не соответствует критерию охвата кода тестами, чтобы результат тестирования коммита (успешно/неуспешно) публиковался с именем ее автора в канале общения, таком как Slack, чтобы воспроизводился громкий звуковой сигнал всякий раз, когда сборка приложения в конвейере CI завершается неудачей, и т. д. Я, как тестировщик в команде, слежу за статусом тестов в CI и за тем, чтобы они исправлялись вовремя. По сути, все эти меры принимаются для оптимизации CI/СТ в команде и поэтому приносят определенные выгоды, но самая важная мера, которая всегда дает положительный результат, — рассказывать команде, не только как, но и почему это делается!

Стратегия непрерывного тестирования

Теперь, познакомившись с процессами и принципами, перейдем к определению и применению стратегий, адаптированных к потребностям вашего проекта.

В предыдущем разделе процесс непрерывного тестирования был продемонстрирован на одном этапе сборки и тестирования, когда все тесты выполняются и выдают результаты в одном цикле. Цикл обратной связи можно ускорить с помощью двух независимых циклов: один применяет тесты (например, все тесты микроуровня) к статическому коду приложения, а другой — тесты макроуровня к развернутому приложению. В каком-то смысле это раннее тестирование: мы используем способность тестов микроуровня (модульных, интеграционных, контрактных) выполняться быстрее, чем тесты макроуровня (API, пользовательские истории, сквозные), для более быстрого получения обратной связи.

На рис. 4.3 показан процесс СТ, состоящий из двух этапов. Как можно видеть, обычная практика состоит в том, чтобы объединить компиляцию приложения с тестами на микроуровне в один этап CI. Традиционно он называется этапом *сборки и тестирования*. Когда команда придерживается принципа построения пирамиды тестирования, как обсуждалось в главе 3, тесты на микроуровне в конечном итоге проверяют широкий спектр функциональных возможностей приложения. В результате этот этап помогает быстро получить обширную информацию о коммите. Этап сборки и тестирования должен выполняться довольно быстро — укладываться

в несколько минут, чтобы в соответствии с рекомендуемыми принципами и правилами команда дождалась его завершения, прежде чем перейти к следующей задаче. Если этот этап продолжается дольше, то команде следует найти способы ускорить его, например распараллелить этапы сборки и тестирования для разных компонентов вместо сборки и тестирования всей базы кода¹.

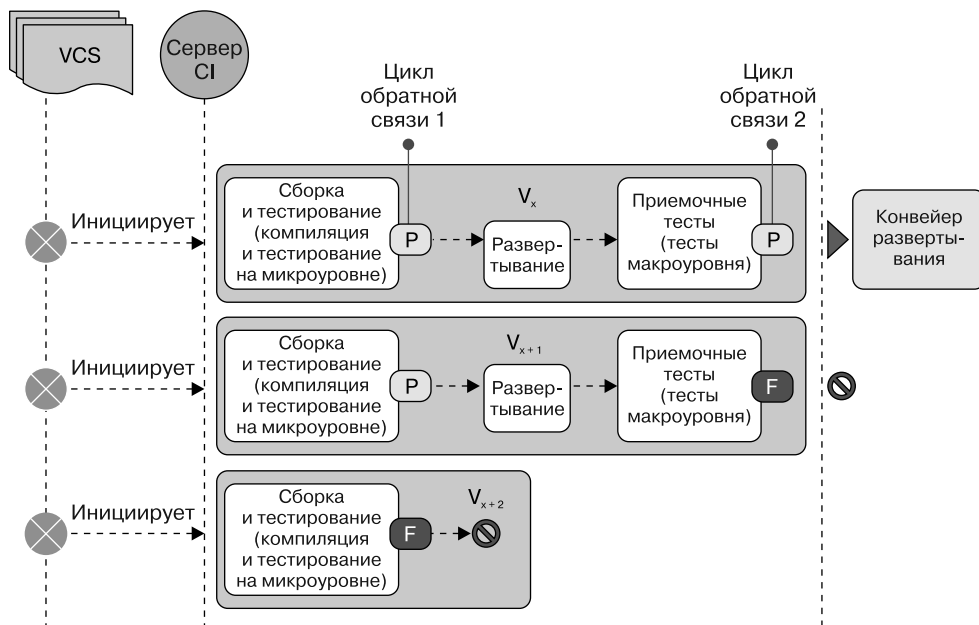


Рис. 4.3. Процесс непрерывного тестирования с двумя циклами обратной связи



В своей книге *Continuous Delivery*² (Addison-Wesley Professional) Джез Хамбл и Дэвид Фарли предлагают сделать этап сборки и тестирования настолько коротким, чтобы он занимал «примерно столько же времени, сколько нужно для приготовления чашки чая, короткого обмена мнениями, проверки электронной почты или разминки».

Как только этапы сборки и тестирования будут пройдены, этап развертывания передаст артефакты приложения в среду CI (иногда ее называют средой разра-

¹ Подробнее об этом и о других принципах работы CI/CD читайте в руководстве *The DevOps Handbook* (IT Revolution Press), составленном Джинном Кимом, Джезом Хамблом, Патриком Дебуа и Джоном Уиллисом. (Ким Д., Дебуа П., Уиллис Д., Хамбл Д. Руководство по DevOps. — М., 2018.)

² Хамбл Д., Фарли Д. Непрерывное развертывание ПО. — М., 2017.

ботки). Следующий этап, называемый *этапом функционального тестирования* или *этапом приемочного тестирования*, запускает тесты макроуровня для проверки приложения, развернутого в среде CI. Только после прохождения этого этапа приложение будет готово к развертыванию в других средах более высокого уровня, таких как среда отдела обеспечения качества, приемочного тестирования на стороне пользователя и промышленной эксплуатации.

Для получения обратной связи на этом этапе может потребоваться больше времени, поскольку приемочные тесты выполняются дольше, а этап запускается после развертывания приложения, что тоже требует времени. Но когда команды правильно реализуют пирамиду тестирования, выполнение двух циклов обратной связи обычно занимает менее часа. Пример, который я привела в главе 3, подтверждает это: когда у команды было около 200 тестов макроуровня, потребовалось 8 часов, чтобы получить обратную связь, но когда они реорганизовали структуру тестирования, приведя ее в соответствие с пирамидой тестирования, то стали тратить всего около 35 минут от принятия решения до готовности к развертыванию, выполняя при этом примерно 470 тестов на микро- и макроуровне.

Еще одно соображение: когда цикл обратной связи короткий, члены команды все равно могут расставить приоритеты проблем, обнаруженных в процессе непрерывного тестирования, для дальнейшего их устранения, даже если взяли в работу новую задачу вскоре после этапа сборки и тестирования. Если на устранение проблем уйдет несколько часов, у них может возникнуть соблазн игнорировать неудачные тесты и отложить их исправление на потом. Это довольно опасно, потому что новый код будет интегрироваться поверх дефектов, при этом он не будет тщательно тестироваться, поскольку решено игнорировать неудачные тесты. Поэтому команда должна продолжать искать и внедрять способы ускорения двух циклов обратной связи, используя такие методы, как распараллеливание выполнения тестов, внедрение пирамиды тестов, удаление повторяющихся тестов и рефакторинг тестов для устранения ожиданий и абстрагирования общих функций¹.

Можно расширить процесс непрерывного тестирования и включить в него получение результатов межфункциональных тестов (рис. 4.4). Команды могут запускать автоматизированные тесты производительности, безопасности и доступности в рамках двух имеющихся циклов обратной связи или настраивать отдельные этапы после этапа приемочного тестирования на сервере CI, достигая цели получения непрерывной быстрой обратной связи о качестве приложения в целом. В следующих главах вы познакомитесь со стратегиями реализации раннего межфункционального тестирования.

¹ Джез Хамбл и Дэвид Фарли подробно обсуждают такие методы оптимизации в книге Continuous Delivery (<https://oreil.ly/continuous-delivery>).

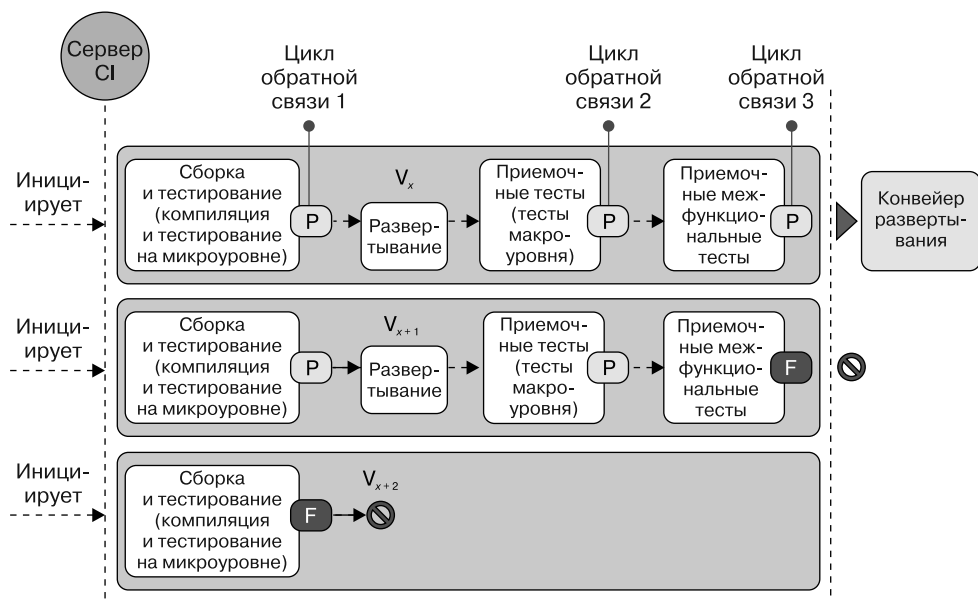


Рис. 4.4. Процесс непрерывного тестирования с тремя циклами обратной связи

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ И НЕПРЕРЫВНОЕ ТЕСТИРОВАНИЕ

Как следует из названия, процесс непрерывной интеграции заканчивается этапом сборки и тестирования. То есть коммит считается интегрированным только тогда, когда он благополучно проходит тесты на микроуровне (по крайней мере, модульные тесты)¹.

Процесс непрерывного тестирования предполагает проверку целостного поведения приложения, включая функциональные и межфункциональные аспекты, для каждого коммита, позволяющую убедиться в его готовности к непрерывной доставке. Фактически непрерывное тестирование не ограничивается выполнением автоматизированных тестов, оно включает также ручное исследовательское тестирование каждого коммита после развертывания. Процесс CI требует от команды автоматизировать сценарии, обнаруженные в ходе исследовательского тестирования, чтобы иметь право называть функциональность или коммит «готовыми».

Когда все тесты запускаются в цепочке конвейеров, для завершения всех этапов может потребоваться довольно много времени и ресурсов. В этом случае можно реорганизовать стратегию процесса непрерывного тестирования — разделить тесты на *дымовые* и *ночные регрессионные* (рис. 4.5).

¹ Более подробную информацию вы найдете в книге Джеза Хамбла, Джина Кима и Николь Форсгрэн *Accelerate* (IT Revolution Press). (Форсгрэн Н., Хамбл Д., Ким Д. Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации. — М., 2022.)

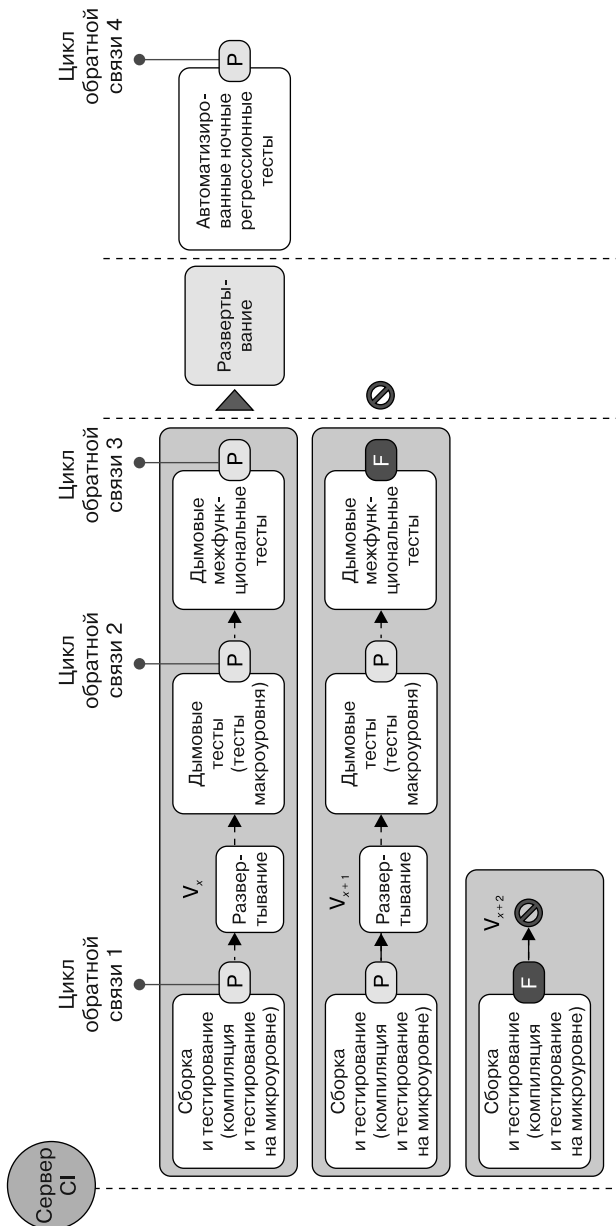


Рис. 4.5. Процесс непрерывного тестирования с четырьмя циклами обратной связи

Термин «дымовое тестирование» был заимствован из мира электротехники. После сборки электрической сети через нее пропускают электрический ток и оценивают, как он проходит. Когда в цепи обнаруживается проблема, идет дым, отсюда и название. Аналогично можно выбрать тесты, проверяющие выполнение потока действий, которые затрагивают каждую функцию в приложении, и сформировать из них пакет дымовых тестов, чтобы запускать их на этапе приемочного тестирования. Используя такие пакеты, можно быстро получить высокоуровневый сигнал о статусе каждого коммита. Как показано на рис. 4.5, коммит готов к развертыванию после этапа дымового тестирования.

Решив провести дымовое тестирование, вам придется дополнить его ночным регрессионным тестированием. Оно настраивается на сервере CI и заключается в запуске всего набора тестов после окончания рабочего дня (например, его можно запланировать на 19:00 ежедневно). Тесты применяются к самой свежей базе кода со всеми коммитами, произведенными в течение рабочего дня. Команда должна выработать привычку анализировать результаты ночного регрессионного тестирования в начале следующего дня и уделять первоочередное внимание исправлению дефектов и сбоев. Иногда это может потребовать изменения сценария тестирования, и эту работу тоже следует классифицировать по приоритетам и запланировать для выполнения в течение дня, чтобы процесс непрерывного тестирования давал правильную обратную связь для предстоящих коммитов.

С помощью этих двух стратегий можно разделить функциональные и межфункциональные тесты. Например, можно запускать нагрузочный тест производительности для одной критической конечной точки после каждого коммита, а остальные тесты производительности — на этапе ночного регрессионного тестирования (тесты производительности обсуждаются в главе 8). Аналогично можно запускать статические тесты сканирования на уязвимости кода на этапе сборки и тестирования, а тесты функционального сканирования на уязвимости (обсуждаемые в главе 7) — на этапе ночного регрессионного тестирования. Недостаток такого подхода заключается в том, что обратная связь задерживается на день. Следовательно, исправление обнаруженных проблем тоже происходит с задержкой. В результате приходится быть осторожными при выборе типов тестов, которые должны запускаться на этапе дымового и ночного регрессионного тестирования. Также обратите внимание на то, что к дымовым тестам относятся только тесты макроуровня и межфункциональные — все тесты на микроуровне по-прежнему должны выполняться на этапе сборки и тестирования.

Чаще всего, когда приложение молодое, можно отказаться от этих стратегий и наслаждаться возможностью запускать все тесты после каждого коммита. Затем, когда приложение начнет расти (как и количество тестов), можно реализовать различные методы оптимизации среды выполнения CI, а в конечном итоге перейти к дымовому и ночному регрессионному тестированию.

Преимущества

Если вас волнует, принесут ли плоды все эти усилия по организации непрерывного тестирования, то взгляните на рис. 4.6, где показаны некоторые преимущества, которые могут мотивировать вас и вашу команду.



Рис. 4.6. Преимущества внедрения процесса непрерывного тестирования

Рассмотрим их по очереди.

Общие цели в области качества

Следование процессу непрерывного тестирования гарантирует, что все члены команды осознают его важность и все вместе работают над достижением целей в области качества с точки зрения как функциональных, так и межфункциональных аспектов, потому что их работа постоянно оценивается в соответствии с этой целью. Это конкретный способ повышения качества.

Раннее обнаружение дефектов

Каждый член команды получает немедленную обратную связь по своим коммитам с точки зрения функциональных и межфункциональных аспектов. Это дает им возможность исправлять проблемы, пока свежи воспоминания, а не возвращаться к коду через несколько дней или недель.

Постоянная готовность к доставке

Поскольку код постоянно тестируется, приложение все время готово к развертыванию в любой среде.

Расширенное сотрудничество

С членами распределенной команды легче сотрудничать, когда они отправляют результаты своего труда в общий репозиторий, и легче выяснять, какой коммит вызвал те или иные проблемы. Благодаря этому снижается вероятность ложных обвинений и уменьшается враждебность.

Общая ответственность за доставку

Ответственность за доставку распределяется между всеми членами команды, а не только между членами команды тестировщиков или старшими разработчиками, потому что каждый отвечает за готовность своих коммитов к развертыванию.

Если у вас есть опыт работы в индустрии ПО, то вы наверняка знаете, как трудно добиться некоторых из этих преимуществ другими способами!

Упражнение

Пришло время взяться за дело. Приведенное здесь пошаговое упражнение покажет, как организовать отправку автоматизированных тестов, созданных в главе 3, в систему VCS, настроить сервер CI и интегрировать с ним автоматизированные тесты, чтобы при каждой отправке порции кода в VCS автоматически выполнялись тесты. В ходе изучения этого упражнения вы научитесь использовать Git и Jenkins.

Git

Разработанная в 2005 году Линусом Торвалдсом, создателем ядра операционной системы Linux, Git является наиболее широко используемой системой управления версиями с открытым исходным кодом. Согласно опросу, проведенному на Stack Overflow в 2021 году (<https://oreil.ly/pb7Pb>), 90 % респондентов применяют Git. Это распределенная система управления версиями, что означает: каждый член команды получает копию всей базы кода вместе с историей изменений. Это обеспечивает командам большую гибкость в плане отладки и независимости в работе.

Установка и настройка

Прежде всего вам понадобится место для размещения своей базы кода. GitHub и Bitbucket — это компании, которые предоставляют облачные хранилища для репозиториях Git (репозиторий — это место хранения базы кода). GitHub позволяет бесплатно размещать публичные репозитории, что делает его популярным, особенно среди сообщества разработчиков ПО с открытым исходным кодом. Итак, для этого упражнения, если у вас еще нет учетной записи GitHub, создайте ее сейчас (<https://github.com/join>).

В своей учетной записи GitHub перейдите в раздел **Your Repositories** ▶ **New** (Ваши репозитории ▶ Создать), чтобы создать новый репозиторий для автоматизированных тестов Selenium. Укажите имя репозитория, например `FunctionalTests`, и сделайте его общедоступным. После успешного создания вы попадете на страницу настройки репозитория. Запишите URL репозитория (https://github.com/<ваше_имя_пользователя>/FunctionalTests.git). На странице имеется набор инструкций по отправке кода в репозиторий с помощью команд Git. Для их запуска вам придется установить и настроить Git на своем компьютере.

Для этого выполните следующие действия.

1. Загрузите и установите Git из командной строки, выполнив следующие команды:

```
// macOS
$ brew install git

// Linux
$ sudo apt-get install git
```

Если вы пользуетесь Windows, загрузите пакет установки для Windows с официального сайта Git (<https://gitforwindows.org>).

2. Проверьте установку, выполнив следующую команду:

```
$ git --version
```

3. Каждый раз, выполняя коммит, вы должны привязать его к имени пользователя и адресу электронной почты для дальнейшего отслеживания. Предоставьте Git эту информацию, выполнив следующие команды, чтобы она автоматически присоединялась при выполнении коммита:

```
$ git config --global user.name "ваше_имя_пользователя"
$ git config --global user.email "ваш_адрес_электронной_почты"
```

4. Проверьте конфигурацию, выполнив следующую команду:

```
$ git config --global --list
```

Рабочий процесс

Процесс работы с Git состоит из четырех этапов (рис. 4.7). Как вы уже знаете, каждый этап имеет свою цель.

Первый этап — ваш *рабочий каталог*, где вы вносите изменения в свой тестовый код (добавляете новые тесты, исправляете тестовые сценарии и т. д.). Второй этап — локальная *промежуточная область*, куда вы добавляете небольшие фрагменты работы, например созданный класс страницы после его завершения. Этот этап позволяет отслеживать вносимые изменения, чтобы вы могли просмотреть и повторно использовать их позже. Третий этап — ваш *локальный репозиторий*. Как упоминалось ранее, Git предоставляет каждому копию всего репозитория вместе

с историей на локальном компьютере. Когда у вас есть рабочая структура тестов, можете выполнить коммит, который переместит в локальный репозиторий все, что вы добавили в промежуточную область. Это позволяет в случае сбоя отозвать обратно весь код как один фрагмент. Наконец, внося все необходимые изменения (в данном случае после завершения тестирования и подготовки к запуску в конвейере CI), можете отправить их в удаленный репозиторий. После этого новый тест будет доступен всем членам вашей команды.

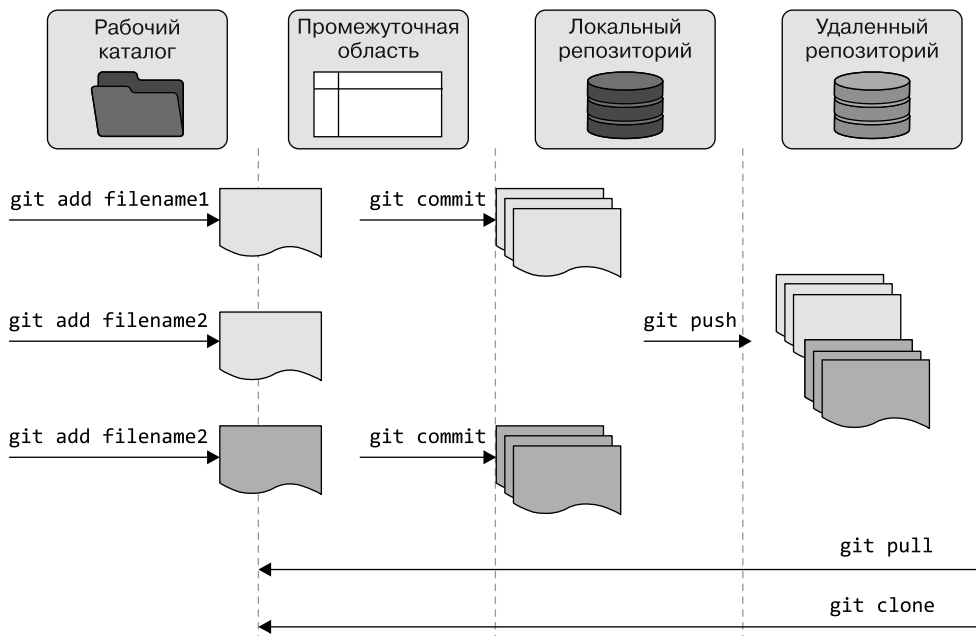


Рис. 4.7. Процесс работы с Git состоит из четырех этапов

Команды Git для перемещения кода по различным этапам показаны на рис. 4.7. Можете попробовать применить их прямо сейчас, как описывается далее.

1. В терминале перейдите в папку, где вы создали автоматизированные тесты Selenium (в главе 3). Выполните следующую команду, чтобы инициализировать репозиторий Git:

```
$ cd /путь/к папке/проекта/
$ git init
```

Эта команда создаст папку `.git` в текущем рабочем каталоге.

2. Добавьте все свои тесты в промежуточную область, выполнив команду:

```
$ git add .
```

В этой команде можно указать определенный файл (или каталог), например:

```
git add имя_файла
```

3. Зафиксируйте изменения в локальном репозитории, добавив осмысленное описание, объясняющее контекст коммита:

```
$ git commit -m "Adding functional tests"
```

Шаги 2 и 3 можно объединить, добавив необязательный параметр, например -a:

```
git commit -am "описание"
```

4. Чтобы отправить свой код в общедоступный репозиторий, нужно сначала настроить его местоположение в локальном Git. Сделайте это, выполнив следующую команду:

```
$ git remote add origin
https://github.com/<ваше_имя_пользователя>/FunctionalTests.git
```

5. Следующий шаг — отправить код в публичный репозиторий. При этом вам придется пройти аутентификацию: указать свое имя пользователя на GitHub и личный токен доступа. Личный токен доступа — это кратковременный пароль, требуемый GitHub для всех операций с августа 2021 года по соображениям безопасности. Чтобы получить личный токен доступа, зайдите в свою учетную запись на GitHub, перейдите в **Settings** ▶ **Developer Settings** ▶ **Personal access tokens** (Настройки ▶ Настройки разработчика ▶ Личные токены доступа), щелкните на ссылке **Generate new token** (Создать новый токен) и заполните необходимые поля. Используйте токен при появлении запроса после выполнения следующей команды:

```
$ git push -u origin master
```



Если вы не хотите аутентифицироваться при каждом взаимодействии с общедоступным репозиторием, то настройте механизм аутентификации через SSH (<https://oreil.ly/Yu10Q>).

6. Откройте свою учетную запись на GitHub и проверьте репозиторий.

Работая в команде, вам придется переносить код своих коллег на свою машину из общедоступного репозитория. Это можно сделать, выполнив команду `git pull`. Если у вас уже есть репозиторий функциональных тестов для вашей команды, то вместо `git init` используйте команду `git clone URL_репозитория`, чтобы получить копию локального репозитория.

В Git есть еще ряд команд, таких как `git merge`, `git fetch` и `git reset`, которые могут облегчить вашу жизнь. Познакомиться с ними поближе можно в официальной документации (<https://git-scm.com/docs>).

Jenkins

Следующий шаг — настройка сервера Jenkins CI на локальном компьютере и интеграция автоматических тестов из вашего репозитория Git.



Цель этой части упражнения — показать, как можно реализовать непрерывное тестирование на практике с помощью инструментов CI/CD, а не научить вас применению технологий DevOps. Команды могут привлекать разработчиков со специальными навыками DevOps или иметь особого человека для управления созданием и обслуживанием конвейеров CI/CD/CT. Однако знать процесс CI/CD/CT и принцип его работы должны и разработчики, и тестировщики, поскольку они будут взаимодействовать с ним и устранять возникающие в нем сбои. Кроме того, с точки зрения тестирования очень важно уметь адаптировать процесс CT к конкретным потребностям проекта и обеспечить правильную цепочку этапов тестирования в соответствии со стратегией CT.

Установка и настройка

Jenkins — это сервер CI с открытым исходным кодом. Чтобы использовать его, загрузите установочный пакет для вашей ОС (<https://oreil.ly/pa0yJ>) и выполните стандартную процедуру установки. После этого запустите службу Jenkins. В macOS установить и запустить службу Jenkins можно с помощью команд `brew`:

```
$ brew install jenkins-lts
$ brew services start jenkins-lts
```

После успешного запуска службы откройте веб-интерфейс Jenkins по адресу: `http://localhost:8080/`. В нем выполните следующие действия по настройке.

1. Разблокируйте Jenkins с помощью уникального пароля администратора, созданного в процессе установки. Веб-страница покажет вам путь к местоположению этого пароля на вашем локальном компьютере.
2. Загрузите и установите часто используемые плагины Jenkins.
3. Создайте учетную запись администратора. Вы будете применять ее для входа в Jenkins.

После первоначальной настройки вы попадете на страницу панели управления Jenkins (рис. 4.8).



В этом упражнении вы настроите сервер CI на своем локальном компьютере, но на практике сервер CI будет размещаться либо в облаке, либо на виртуальной машине в той же сети, что и компьютеры вашей команды, чтобы все ее члены могли получить к нему доступ.

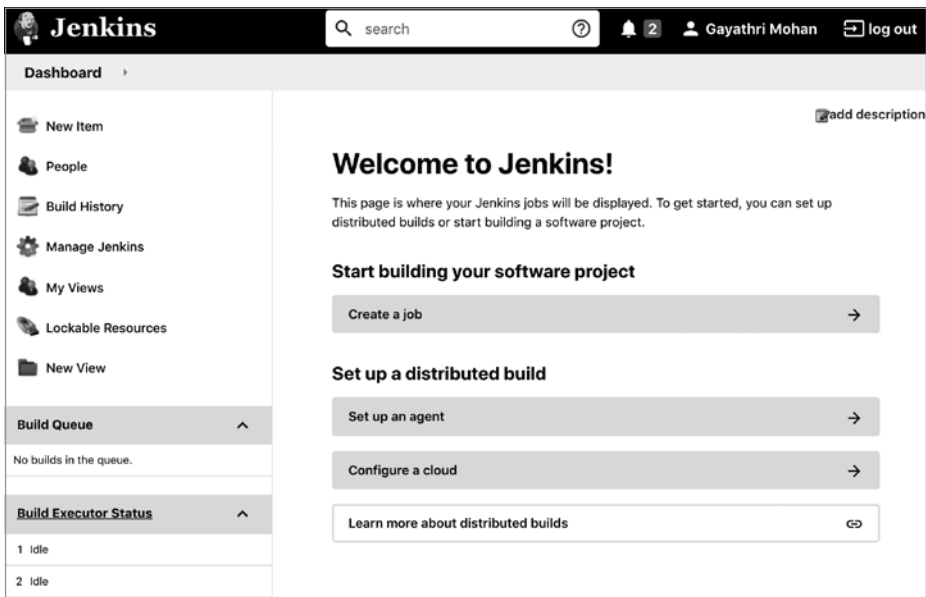


Рис. 4.8. Панель управления Jenkins

Рабочий процесс

Чтобы настроить новый конвейер для автоматизированных тестов, выполните следующие действия.

1. На панели управления Jenkins перейдите в раздел **Manage Jenkins** ► **Global Tool Configuration** (Управление Jenkins ► Глобальная конфигурация инструмента), чтобы настроить переменные среды `JAVA_HOME` и `MAVEN_HOME`, как показано на рис. 4.9 и 4.10 соответственно. Вы можете ввести команду `mvn -v` в терминале, чтобы получить оба местоположения.

Рис. 4.9. Настройка `JAVA_HOME` в Jenkins

Maven installations	
Add Maven	
Maven	
Name	maven 3.6.1
MAVEN_HOME	/usr/local/Cellar/maven/3.6.1/libexec
<input type="checkbox"/> Install automatically	

Рис. 4.10. Настройка MAVEN_HOME в Jenkins

- Вернувшись на панель управления, выберите слева пункт New Item (Новый элемент), чтобы создать новый конвейер. Введите имя конвейера, например **Functional Tests**, и выберите параметр **Freestyle project** (Произвольный проект). Вы попадете на страницу конфигурации конвейера (рис. 4.11).

General | Source Code Management | Build Triggers | Build Environment | Build | Post-build Actions

Description: Continuous Testing Exercise

[Plain text] Preview

☐ Discard old builds

☒ GitHub project

Project url: https://github.com/username/FunctionalTests/

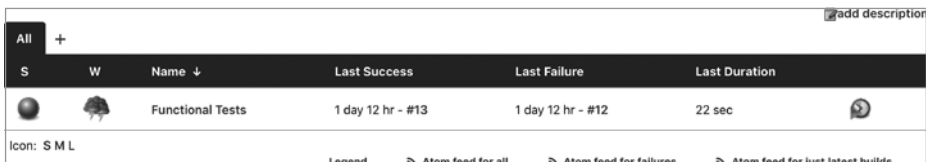
Advanced...

Рис. 4.11. Страница настройки конвейера в Jenkins

- Чтобы настроить конвейер, введите следующие данные.
 - На вкладке **General** (Общие) добавьте описание конвейера. Выберите **GitHub project** (Проект GitHub) и введите URL своего репозитория (без расширения **.git**).
 - На вкладке **Source Code Management** (Управление исходным кодом) выберите **Git** и введите URL своего репозитория (на этот раз с расширением **.git**). Jenkins будет использовать этот URL для клонирования репозитория командой **git clone**.
 - Вкладка **Build Triggers** (Триггеры сборки) предоставляет несколько параметров, настраивающих, когда и как автоматически запускать конвейер. Например, параметр **Poll SCM** (Опрос SCM) можно использовать для того,

чтобы каждые 2 мин обращаться к репозиторию Git, проверяя наличие новых изменений, и при обнаружении таковых запускать тестирование. Параметр **Build Periodically** (Периодическая сборка) можно задействовать для планирования запуска тестов через фиксированные интервалы даже при отсутствии новых изменений в коде. Эту возможность можно использовать для настройки ночных регрессионных тестов. Аналогично параметр **GitHub hook trigger for GITScm polling** (Триггер GitHub для опроса GITScm) настраивает плагин GitHub для отправки триггера в Jenkins при появлении новых изменений. Для простоты выберите **Poll SCM** (Опрос SCM) и, чтобы организовать опрос репозитория функциональных тестов каждые 2 мин введите следующее значение: `H/2 * * * *`.

- Поскольку наш фреймворк функционального тестирования Selenium WebDriver использует Maven, выберите параметр **Invoke top-level Maven targets** (Вызов целей Maven верхнего уровня) на вкладке **Build** (Сборка). Выберите локальную установку Maven, которую вы настроили в главе 3. В поле **Goals** (Цели) введите этап жизненного цикла Maven, который должен выполняться конвейером: `test`. Он будет выполнять команду `mvn test` в каталоге проекта.
 - На вкладке **Post-build Actions** (Действия после сборки) можно объединить несколько конвейеров, то есть запустить конвейер межфункциональных тестов после выполнения конвейера функциональных тестов и создать полный конвейер CD¹.
4. Сохраните настройки и перейдите на панель управления. Вы увидите только что созданный конвейер (рис. 4.12).



S	W	Name ↓	Last Success	Last Failure	Last Duration
		Functional Tests	1 day 12 hr - #13	1 day 12 hr - #12	22 sec

Icon: S M L

Legend Atom feed for all Atom feed for failures Atom feed for just latest builds

Рис. 4.12. Вновь созданный конвейер на панели управления Jenkins

5. Щелкните на имени конвейера на панели управления и после перехода на страницу этого конвейера выберите параметр **Build Now** (Собрать сейчас) на панели слева. Конвейер клонирует репозиторий на вашем локальном компьютере и выполнит команду `mvn test`. В процессе тестирования вы увидите, как браузер Chrome будет открыт и закрыт несколько раз.
6. Найдите папку **Workspace** на той же странице. В ней вы увидите локальную клонированную копию кода из репозитория и отчеты с результатами тестирования. Их можно использовать для отладки.

¹ Дополнительную информацию о работе с конвейерами ищите в документации Jenkins (<https://oreil.ly/iYASL>).

- Внизу на панели слева на той же странице выберите текущее количество сборок, выполненных конвейером. Вам будет предоставлена возможность просмотреть вывод в консоль на левой панели. В этом представлении будут показаны действия, выполняемые в реальном времени, что может пригодиться для отладки.

Поздравляю, настройка CI завершена!

Аналогично нужно добавить этапы соответствующих тестов (статических, приемочных, дымовых, межфункциональных) в соответствии со своей стратегией непрерывного тестирования, чтобы завершить комплексную настройку CD для проекта. Убедитесь, что этапы запускаются не только после отправки изменений кода приложения, но и после изменений конфигурации, инфраструктуры и тестового кода!

Четыре ключевых показателя

Конечный результат всех этих усилий по настройке процессов CI/CD/CT (и соблюдению принципов и правил, изложенных ранее) — команда, квалифицирующаяся как элитная или высокопроизводительная в соответствии с четырьмя ключевыми показателями (4КП), предложенными командой Google DevOps Research and Assessment (DORA). Команда DORA сформулировала 4КП, опираясь на обширные исследования (<https://oreil.ly/bDj9t>), и описала, как использовать эти показатели для количественной оценки качественного уровня команды разработчиков: элитный, высокий, средний или низкий. Книга *Accelerate* Джеза Хамбла, Джина Кима и Николь Форсгрэн — отличный источник для изучения деталей исследования.

Проще говоря, четыре следующих ключевых показателя дают возможность измерить темп разработки ПО и стабильность релизов.

Время выполнения

Время от коммита кода до его готовности к развертыванию в промышленной среде.

Частота развертывания

Частота, с которой программное обеспечение развертывается в промышленной среде или в магазине приложений.

Среднее время восстановления

Время, необходимое для восстановления после любых сбоев в работе или обслуживании.

Процент неудачных изменений

Процент изменений, развернутых в промышленной среде, которые требуют последующего исправления, например отката к предыдущей версии, или приводят к ухудшению качества обслуживания.

Первые два показателя — время выполнения и частота развертывания — отражают темп работы команды. Они измеряют, как быстро команда может принести пользу конечным пользователям и как часто она добавляет новые ценности для них. Однако, стремясь стать полезной клиентам, команда не должна идти на компромиссы в отношении стабильности ПО. Эту характеристику оценивают два следующих показателя. Среднее время восстановления и процент неудачных изменений оценивают стабильность выпускаемого ПО. В современном мире сбои программного обеспечения неизбежны, и эти показатели сообщают, насколько быстро команда восстанавливает нормальное функционирование своего ПО после сбоев и как часто такие сбои происходят из-за новых выпусков. Как видите, все вместе 4КП дают четкое представление о работе команды разработчиков, измеряя их скорость, реактивность и способность обеспечивать качество и стабильность.

Целевые значения показателей для получения статуса элитной команды, определенные в результате исследований DORA, представлены в табл. 4.1.

Таблица 4.1. Четыре ключевых показателя элитной команды

Показатель	Цель
Частота развертывания	По требованию (до нескольких раз в день)
Время выполнения	Меньше одного дня
Среднее время восстановления	Меньше одного часа
Процент неудачных изменений	0–15 %

Как обсуждалось ранее, одно из основных преимуществ строгого следования процессу CI/CD/CT — возможность доставлять пользу клиентам по первому требованию. Точно так же, размещая автоматизированные тесты на нужных уровнях приложения, можно протестировать код в рамках процесса непрерывного тестирования и подготовить его к развертыванию в течение нескольких часов (то есть время выполнения будет меньше одного дня).

Кроме того, поскольку тесты для проверки функциональных и межфункциональных требований автоматизированы и выполняются как часть процесса CT, не должно возникнуть проблем с удержанием процента неудачных изменений в рекомендуемом диапазоне 0–15 %. Таким образом, усилия, приложенные вами к настройке процессов CI/CD/CT, позволят вашей команде заработать элитный статус согласно определению DORA. Исследование DORA (<https://oreil.ly/lvf0X>) также показывает, что элитные команды способствуют успеху организации с точки зрения прибыли, цены на акции, удержания клиентов и других критериев. А когда у организации дела идут хорошо, она заботится о своих сотрудниках, верно?

Ключевые выводы

- Процесс непрерывного тестирования автоматически проверяет качество приложения с точки зрения функциональных и межфункциональных аспектов после каждого изменения.
- Непрерывное тестирование во многом зависит от процесса непрерывной интеграции. А вместе они обеспечивают возможность непрерывной доставки ПО клиентам.
- Чтобы процессы непрерывной интеграции и тестирования приносили выгоду, команды должны следовать строгим принципам и правилам.
- Спланируйте процесс непрерывного тестирования так, чтобы постоянно получать быструю обратную связь в нескольких циклах.
- Преимущества непрерывного тестирования многочисленны, и многих из них, включая общие цели в области качества, общую ответственность за доставку и улучшенное сотрудничество внутри распределенных команд, было бы трудно получить иным способом.
- Обычно настройкой и обслуживанием CI/CD занимаются инженеры DevOps, но за разработку стратегии непрерывного тестирования и правильную работу циклов обратной связи отвечают тестировщики команды. Они должны внимательно следить за практикой СТ в команде и гарантировать получение пользы от усилий, затраченных на создание и поддержку тестов.
- Строго следуя процессам CI/CD/СТ, ваша команда станет элитной в том смысле, как это определено исследованием DORA. А элитная команда способствует успеху всей организации!

Тестирование данных

Создайте или разрушите доверие к данным!

Найдите минутку и подумайте об онлайн-сервисах, которыми пользуетесь каждый день. Вы обнаружите, что, по сути, они предлагают всего два вида услуг: отдают вам свои данные или принимают ваши данные и обрабатывают их от вашего имени. Например, интернет-магазин, заказ такси, доставка еды, потоковое воспроизведение фильмов и онлайн-игры — это примеры сервисов первой категории, основная ценность которых заключается в имеющихся у них данных, тогда как приложение для заметок, приложения социальных сетей, таких как Facebook, Twitter и Instagram, сайты блогов и подобные им живут за счет сбора ваших данных! В обоих случаях данные находятся в центре внимания, а вокруг них вращаются функциональные возможности, дизайн пользовательского интерфейса, брендинг и маркетинг. Поясню это на примере: Amazon по своей сути является бизнесом по обработке данных. Его фундаментом является сбор информации о товарах, а на нем строятся основные функции, такие как покупка и доставка товаров. Брендинг и маркетинг компании ненавязчиво указывают на превосходство данных: логотип Amazon со стрелкой между буквами A и Z сообщает миру, что компания располагает огромным разнообразием данных о товарах от A до Z.

Данные бесценны для любого приложения, и если их целостность не поддерживается должным образом, то доверие клиентов к приложению может быстро исчезнуть, а вместе с ним упадут продажи и бизнес в целом. Представьте, что вы перевели деньги между двумя своими счетами через приложение онлайн-банкинга, и хотя приложение сообщило об успешном выполнении транзакции, балансы на обоих счетах в течение некоторого времени отражают неправильные суммы. Почти наверняка вы запаникуете и начнете сомневаться в честности банка! Такая реакция возникает даже в ходе работы с приложением, обрабатывающим не особенно важные данные. Допустим, сообщения, публикуемые вами на сайте блога, не отображаются у ваших коллег или сайт социальной сети теряет ваши семейные фотографии. Вне всяких сомнений, любой из нас был бы разочарован, потому что наши данные важны для нас независимо от их относительной значимости! В итоге такие проблемы заставят нас искать альтернативы.

Из этих примеров можно сделать вывод, что целостность данных чрезвычайно важна, соответственно, тестирование хранения, обработки и представления данных

имеет решающее значение для успеха приложения. В этой главе мы обсудим основы такого тестирования. Сначала вы познакомитесь с разными способами хранения и обработки данных, в частности с базами данных, кэшами, системами потоковой передачи и пакетной обработки. Знакомство с этими системами позволит выявить новые тестовые сценарии, характерные для каждой из них, и особенно случаи сбоев, вызванные проблемами с параллелизмом, распределенной обработкой данных и асинхронной связью. Во второй части главы мы разберем упражнения, которые помогут вам организовать автоматизированное и ручное тестирование данных с использованием различных инструментов.

ТЕСТИРОВАНИЕ ДАННЫХ И ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Кто-то может заявить, что тестирование данных должно рассматриваться как часть тестирования функциональных возможностей приложения, и отчасти он прав. Однако если подумать об одной и той же функциональности с позиции потоков данных, то можно обнаружить новые тестовые сценарии, которым посвящена эта глава.

Кроме того, тестирования функциональности через пользовательский интерфейс и API может оказаться недостаточно. Вам также может понадобиться отдельно проверить целостность данных в системах хранения и обработки, чтобы убедиться в полноте функциональности. А для этого необходимо освоить определенный набор инструментов и методов. Кроме того, здесь вы увидите, что природа этих систем хранения и обработки данных требует новых тестовых сценариев, поэтому нужны специальные знания о системах данных. Навыки тестирования данных охватывают все это.

Другими словами, чтобы полностью протестировать функциональность, необходимы также навыки тестирования данных.

Введение

Начнем со знакомства с системами хранения и обработки данных, которые обычно используются в мобильных и веб-приложениях, чтобы понять особенности тестирования каждой из них. И снова в качестве примера возьмем простое приложение электронной коммерции, рассмотренное в главе 3. На рис. 5.1 показана архитектура этого приложения с выделенной системой хранения и обработки данных.

Как вы наверняка помните, приложение имеет уровень пользовательского интерфейса, который взаимодействует с набором сервисов, решающих разные бизнес-задачи. Сервисы, в свою очередь, подключены к централизованной базе данных, где хранятся все данные приложения. На рис. 5.1 можно заметить еще три системы обработки данных: сервер кэша, систему пакетной обработки и поток событий. Стрелки показывают, как данные передаются между ними. Давайте проследим поток данных, начав с уровня пользовательского интерфейса, чтобы понять, какие роли играют эти системы данных.

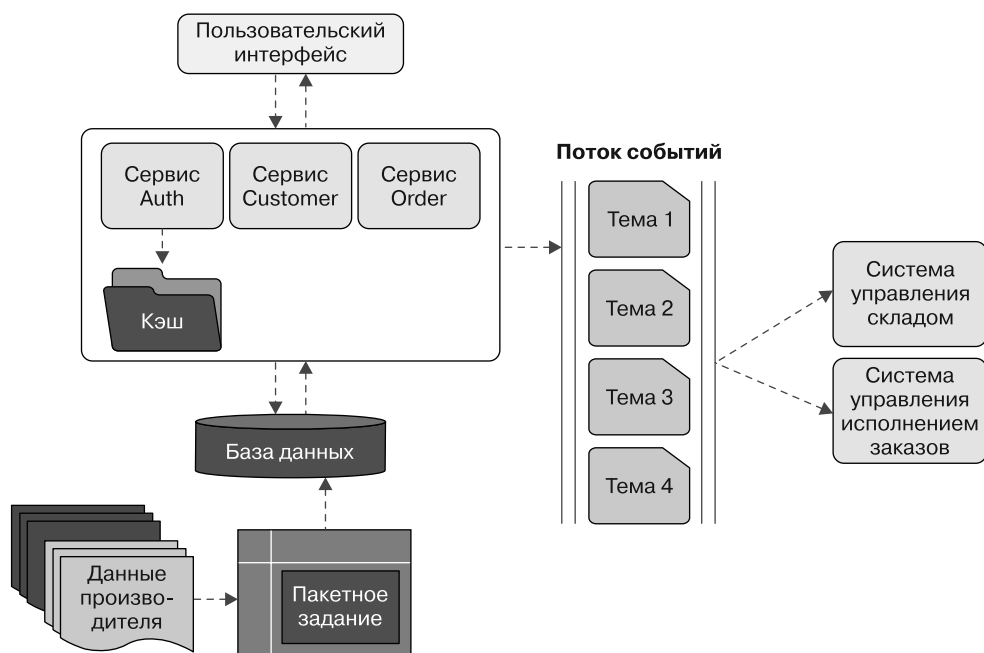


Рис. 5.1. Простое приложение электронной коммерции с четырьмя системами хранения и обработки данных

Для начала предположим, что конечный пользователь пытается войти в приложение, вводя свои учетные данные. Уровень пользовательского интерфейса передает учетные данные сервису аутентификации Auth, а тот, в свою очередь, — в базу данных для проверки подлинности. Если введенные учетные данные совпадают с хранимыми, то служба аутентификации (предположим, что она действует в соответствии с протоколом OAuth 2.0, <https://oreil.ly/FgEwf>) возвращает токен доступа и сохраняет его в кэше. Это ключевой фрагмент внутренних данных приложения, потому что любой последующий запрос от пользователя будет сопровождаться этим токеном доступа, доказывающим, что запрос отправлен конечным пользователем, подтвердившим свою личность.

Пойдем дальше и представим, что пользователь пытается разместить заказ из пользовательского интерфейса. Уровень пользовательского интерфейса создает запрос для отправки сервису управления заказами Order и добавляет токен доступа в заголовок запроса. Служба управления заказами связывается со службой аутентификации, чтобы убедиться, что токен доступа действителен, а служба аутентификации, в свою очередь, запрашивает кэш. Если срок действия токена доступа уже истек, то кэш автоматически удалит его и сервисы вернут код состояния 404. Получив ответ 404, пользовательский интерфейс перенаправит пользователя обратно на страницу входа.

Если, напротив, токен доступа действительный (то есть срок его действия не истек), то сервис аутентификации подтверждает это и сервис управления заказами создает заказ в базе данных. Сервис Order также создаст событие с информацией о новом заказе и поместит его в систему потоковой передачи событий, чтобы действующие далее системы — управления складом и управления исполнением могли решать свои задачи. Следует отметить, что ответственность сервиса управления заказами заканчивается размещением события в системе потоковой передачи событий, и ее не волнует, правильно ли выполняют свою работу действующие далее системы. Эти системы постоянно опрашивают систему потоковой передачи событий и извлекают только те события, которые имеют к ним отношение. Например, событие изменения адреса клиента может быть интересно системе управления исполнением, но системе управления складом оно не дает никакой полезной информации, а вот событие создания заказа может быть интересно обеим системам.

Кроме прочего, существует система пакетной обработки для анализа сведений о товарах различных производителей. Она запускается автоматически в запрограммированное время — скажем, один раз в день в полночь — и импортирует все новые и обновленные данные в базу данных, чтобы новые товары и любые другие изменения стали доступны пользователям на следующий день.

Как показывает этот пример, каждая из четырех систем обработки данных играет важную роль в удовлетворении основных требований приложения. Давайте рассмотрим их более подробно и выясним, какие уникальные свойства делают их подходящими для применения в соответствующих контекстах и какие новые тестовые сценарии они предоставляют.

Базы данных

Базы данных не нуждаются в особом представлении, поскольку они являются распространенным компонентом практически всех приложений и представляют собой устоявшуюся систему хранения данных. Одна из причин такой широкой распространенности — высокая надежность хранения данных, поскольку они хранятся на жестком диске и теряются только в случае серьезных аппаратных сбоев.

Те, кто не знаком с базами данных, могут представить их в виде шкатулок для драгоценностей, где в соответствующих отделениях хранятся различные украшения, которые можно достать при необходимости. Каждое украшение хранится в шкатулке до тех пор, пока его не заберут или не заменят. Данные приложения тоже организованы и хранятся в базах данных, и их можно запрашивать при необходимости. Приложение может *создавать* (create) новые данные, а также *читать* (read), *изменять* (update) или *удалять* (delete) существующие данные в соответствии с требованиями функциональности (эти четыре основные операции обычно обозначаются аббревиатурой CRUD).

В зависимости от способа организации данных, например в виде таблиц, документов JSON или XML или графов, базы данных можно разделить на реляционные, документальные и графовые соответственно¹. Реляционные базы данных являются доминирующей категорией, и в последние несколько десятилетий они прекрасно удовлетворяли широкий спектр требований к хранению данных, предъявляемых приложениями. MySQL и PostgreSQL — примеры реляционных баз данных с открытым исходным кодом, в упражнениях этой главы мы рассмотрим работу с базой данных PostgreSQL.

В реляционной БД данные хранятся в табличных структурах со строками и столбцами. Каждая строка таблицы представляет набор связанной информации, разделенной на столбцы, как в таблице *Customers*, показанной в табл. 5.1.

Таблица 5.1. Пример табличной структуры в реляционной базе данных

UUID (первичный ключ)	Имя клиента (varchar 30)	Телефон (int)	Электронный адрес (varchar 254)	Адрес доставки (varchar 100)
019367	Alice	4567879	alice@xyz.com	8/13, Block A
045678	Bob D'Arcy	0898678	bobdarcy@xyz.com	23-A, Winscent Square

Столбцы могут иметь predetermined имена и свойства, такие как тип данных и максимальная длина. Кроме того, каждой строке присваивается универсальный уникальный идентификатор (Universally Unique Identifier, UUID), который служит для связи записей в нескольких таблицах. Например, список клиентов в нашем приложении может храниться в таблице, где каждая строка представляет сведения об одном клиенте, такие как его имя, адрес электронной почты, номер телефона и адрес доставки. В этом случае уникальный идентификатор записи будет играть роль уникального идентификатора пользователя и храниться вместе с ним. Впоследствии его можно будет применять, запрашивая информацию о клиенте. Тот же идентификатор пользователя можно задействовать в других таблицах, например в таблице истории учетных записей, что позволяет получить всю информацию о любом конкретном пользователе. Такое определение таблиц, строк, имен столбцов, уникальных идентификаторов и так далее называется *схемой базы данных*. Схема базы данных создается разработчиками или администратором БД на основе особенностей использования приложения. Схема может быть переопределена в ходе поставки и по мере роста требований приложения. Для выполнения этих операций в реляционных базах данных применяется предметно-ориентированный язык, называемый языком структурированных запросов (Structured Query Language, SQL, произносится «эс-кю-эль»).

¹ Обзор разных моделей данных и языков запросов вы найдете в главе 2 книги Мартина Клеппмана *Designing Data-Intensive Applications* (O'Reilly) (<https://oreil.ly/T3ZQj>). (Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2019.)

Учитывая все сказанное, у нас может появиться желание протестировать следующие основные тестовые сценарии.

- Проверка позитивного тестового сценария, когда информация, полученная от пользователя через пользовательский интерфейс, должна сохраняться и соответствующим образом связываться в базе данных.
- Проверка граничных значений для разных типов столбцов и длин входных данных. Например, если в базе данных поле имени клиента может хранить не более 20 символов, то аналогичное ограничение должно быть применено в пользовательском интерфейсе и при его превышении пользователь должен получать сообщение об ошибке.
- Тестирование с помощью входных данных, содержащих элементы языка SQL. Например, может ли Боб Д'Арси, чье имя включает апостроф, правильно сохранить свое имя в БД? Или его нужно как-то предварительно преобразовать?
- Что случится, если во время операции записи произойдет внезапный сбой в сети? Запишутся ли данные лишь в некоторые таблицы из тех, в которые должны записаться? Этот сценарий усложняется, когда операция записи распределяется между несколькими сервисами.
- Как в таких случаях повлияет на результат повторная попытка выполнить операцию?
- Какова длительность тайм-аута перед тем, как приложение повторит попытку выполнить операцию с базой данных, и каков будет процесс взаимодействия с пользователем?

Когда появляется фактор параллелизма, то есть когда несколько пользователей и систем одновременно обращаются к базе данных для чтения и записи, следует подумать о дополнительных тестовых сценариях, в частности проверяющих состояние гонки. Вот несколько соображений, которые помогут запустить мыслительный процесс.

- Действия одного пользователя могут конфликтовать с действиями другого и приводить к потере изменений. Например, когда два пользователя покупают один и тот же товар в один и тот же момент, количество товара может уменьшиться только на одну единицу, а не на две.
- Аналогично пользователи могут увидеть несовпадающие данные, если приложение прочитает лишь часть изменений. Предположим, что запас некоторых недоступных товаров пополняется и процесс сначала меняет флаг доступности товара на `true` в одной таблице, а затем обновляет количество доступных товаров в другой. Между этими двумя операциями конечный пользователь может видеть товар доступным, но с нулевым количеством.
- Параллелизм может влиять также на общие ресурсы. Например, если два пользователя одновременно покупают последний доступный товар, задействуя

вариант оплаты наложенным платежом, то возможно, что товар может быть связан с одним пользователем, а счет на оплату — с другим.

- Кроме того, параллелизм накладывает ограничения на производительность базы данных. Поэтому тестирование производительности с ожидаемыми объемами данных приобретает особую важность.

Следует отметить, что тестовые сценарии, связанные с параллелизмом, трудно моделировать и знать их полезно в основном на этапе анализа, чтобы на них можно было опираться во время разработки.

Помимо одновременного доступа к одному экземпляру, базы данных обеспечивают масштабируемость посредством *репликации*. Под репликацией понимается создание избыточных экземпляров (реplik) одних и тех же данных. Экземпляры обычно хранятся отдельно друг от друга, в различных географических точках, чтобы повысить производительность для пользователей в разных местах, скажем на Восточном и Западном побережье США или в Северной Америке и Европе. В таких случаях необходим механизм, позволяющий синхронизировать все реплики с последними обновлениями. Обычно для этого одна из реплик назначается *лидером* и ответственной за отправку обновлений другим репликам — *последователям*. Такая ситуация может привести к задержке репликации, когда последователям нужно какое-то время, чтобы получить обновление и перейти в то же состояние, что и у лидера. Задержка может составлять от нескольких секунд до нескольких минут в зависимости от задержки в сети, трафика к этому экземпляру и т. д. Эта модель достижения согласованного состояния через определенный период времени называется *конечной согласованностью* (eventual consistency).



Узнать больше о других моделях согласованности можно в руководстве Дженсена с интерактивной картой (<https://jepson.io/consistency>).

Модель конечной согласованности хорошо подходит для таких приложений, как Twitter или Facebook, где некоторое отставание в показе самых последних публикаций не оказывает большого влияния на пользователей. Однако в некоторых других приложениях задержка может сбить с толку и даже подорвать доверие к ним. Давайте обсудим некоторые возможные проблемы.

Чтение собственных изменений

Предположим, пользователь обновляет информацию в своем профиле, а затем, стремясь убедиться, что изменения сохранились, снова открывает страницу профиля через несколько секунд. Обновления могут не успеть распространиться по всем ведомым репликам, поэтому, если приложение читает данные из такой отстающей реплики, то пользователь может увидеть старую информацию

в своем профиле. Удивленный этим, он может повторно ввести изменения. Если этот цикл повторится несколько раз, то, помимо разочарования пользователя, система может оказаться перегруженной, что приведет к увеличению задержки репликации.

Путешествие во времени

Допустим, пользователь следит за оперативной информацией о соревнованиях по крикету на спортивном веб-сайте. Он обновляет страницу каждые несколько секунд, чтобы как можно скорее увидеть результаты. Если веб-сайт читает данные из нескольких ведомых реплик с конечной согласованностью, то у пользователя может возникнуть ощущение путешествия во времени. Например, при первом обновлении счет может отображаться как 116 ранов (пробежек) за 5 оверов (периодов), а при следующем обновлении веб-сайт может прочитать данные из отстающей реплики и показать счет как 110 ранов за 4,5 овера.

Непоследовательный порядок

Иногда данные объединяются последовательно и могут терять смысл, если последовательность нарушается. Например, в обсуждении публикации в Facebook должен поддерживаться порядок, в котором пользователи пишут свои комментарии, но, когда задержки репликации не учитываются и не обрабатываются должным образом, пользователь может увидеть комментарий, отвечающий на вопрос, не видя перед ним самого вопроса.

Конфликты записи

Чтобы избежать появления единственной точки отказа, иногда для управления репликацией назначают несколько ведущих реплик. В таких случаях обновления могут отправляться разным ведущим репликам, что способно привести к конфликтам записи. Конфликты записи возникают всякий раз, когда один ресурс изменяется несколькими сторонами, как, например, презентация Google, одновременно редактируемая несколькими членами команды. В таком случае разные правки одного и того же текста могут быть приняты разными ведущими репликами, но при объединении обновлений возникнет конфликт, какую из них считать окончательной.

Самое замечательное, что решения этих распространенных проблем хорошо известны и многие базы данных сами решают их. Однако осознание таких проблем и бдительность в отношении возможности их появления крайне важны как при разработке приложений, так и при тестировании.

В заключение отмечу следующее: при тестировании баз данных учитывайте особенности данных вашего приложения, а также потенциальные проблемы, такие как сбои сети, конфликты параллелизма и другие проблемы с распределенными данными.

Кэши

Кэш — это область в памяти, где хранятся данные в виде пар «ключ/значение». Хранение данных в памяти повышает производительность на несколько порядков, поскольку приложению не приходится обращаться к тяжеловесной серверной системе хранения, такой как традиционная реляционная база данных. Популярные сегодня инструменты кэширования, такие как Memcached и Redis (<https://oreil.ly/oqCZw>), могут хранить терабайты данных и возвращать ответ менее чем за миллисекунду. Но с точки зрения долговечности базы данных имеют преимущество, поскольку данные записываются на диск.



Redis эволюционировал и теперь предоставляет множество дополнительных функций, помимо простого кэширования данных в памяти. Его даже можно настроить для сохранения на диске снимка данных на определенный момент времени для поддержки возможности восстановления. Чтобы узнать о Redis больше, загляните в официальную документацию (<https://redis.io/topics/introduction>).

Учитывая эти плюсы и минусы, кэшировать рекомендуется только те данные, которые по своей природе являются временными и часто необходимы приложению. Например, в нашем приложении электронной коммерции в кэше сохраняются токены доступа, потому что они, как предполагается, имеют короткий срок жизни (пока пользователь не войдет в систему) и приложение должно часто обращаться к ним изнутри для проверки подлинности каждого запроса на обслуживание. Кроме того, утрата всех токенов в случае сбоя кэша повлияет на процесс минимально: текущим пользователям просто придется выполнить вход еще раз (возникающее при этом неудовольствие едва ли можно сравнить с огорчением от потери ценных личных данных). Кэш идеально подходит для такого сценария, не требующего высокой надежности, такой как в базе данных.

Альтернативный и распространенный подход — реплицировать часто используемые данные приложения как в кэше, так и в базе данных. В таких случаях код приложения должен нести ответственность за поддержание жизненного цикла кэшированных данных: синхронизацию их с БД, очистку старых данных, возврат к БД в случае сбоя кэша и т. д. Эти ситуации станут тестовыми сценариями, когда данные будут реплицироваться в базе данных и кэше. Вот еще несколько тестовых сценариев.

- Для данных в кэше устанавливается предельное время хранения, по истечении которого они становятся недействительными. Например, можно настроить хранение токенов доступа в течение 30 с. По завершении этого времени мы должны убедиться, что служба аутентификации сгенерирует новый токен и сохранит его в кэше.

- Если кэш становится единственной точкой отказа приложения, как в случае сбоя, из-за которого всем пользователям приходится снова выполнять вход в систему, необходимо протестировать их перенаправление на страницу входа.
- Если экземпляры сервиса реплицируются, то вместе с ними будут реплицироваться и их кэши, в результате чего получится распределенный кэш. В таком случае вам, возможно, придется убедиться, что все функции работают правильно. (Большинство реализаций распределенного кэша, такие как Redis Cluster, изначально поддерживают перенаправление на правильный экземпляр кэша, следовательно, задача тестирования сводится к проверке функционального потока.)
- Тестирование производительности приложения при максимальной нагрузке снова приобретает решающее значение.

Системы пакетной обработки

Система пакетной обработки — это система, в которой программа или задание преобразует входные данные в желаемый результат не по мере их поступления, а пакетами, собранными за определенное время. Пакетные задания могут быть написаны с помощью таких фреймворков и библиотек, как Spring Batch или Apache Spark, и выполняться автономно, без вмешательства пользователя. Входные данные для пакетных заданий могут находиться в файлах, записях базы данных, изображениях и т. д. Объем входных данных может быть огромным, из-за чего задания могут выполняться по нескольку часов или даже дней. Фактически производительность пакетного задания определяется количеством данных, которое оно может обработать за единицу времени, а не временем отклика, как в случае с базами данных или кэшами.

Типичными примерами пакетной обработки могут служить создание отчетов, формирование счетов, составление ежемесячных расчетных ведомостей и подготовка данных перед обучением моделей машинного обучения. В этих примерах можно заметить несколько общих закономерностей: пакетные задания преобразуют неорганизованные или разреженные данные в значимые структуры данных и для бесперебойной работы приложения такие преобразования не обязательно должны выполняться в реальном времени.

Чтобы проиллюстрировать сказанное, вернемся к приложению электронной коммерции. Когда поставщики хотят представить обновленные каталоги своих продуктов, они присылают их в виде файлов. Те могут содержать тысячи записей о товарах, для каждого из которых указываются артикул, цвет, размер, цена и т. д. Ключи в записях могут различаться для разных поставщиков, а файлы могут быть в разных форматах, например JSON или CSV, в зависимости от организации вну-

тренних систем поставщика. Эти разрозненные данные во всех их вариациях нужно свести в общую структуру, используемую приложением. То есть файлы следует преобразовать в записи базы данных, чтобы приложение могло отображать их в пользовательском интерфейсе. Здесь следует отметить, что обновленный каталог не требуется отображать в реальном времени, это можно сделать и на следующий день, и через несколько дней. Следовательно, обрабатывать обновленные каталоги можно в пакетном режиме.

Мы можем написать пакетное задание (или несколько заданий), читающее записи из разных файлов, извлекающее нужную информацию и преобразующее ее в записи базы данных. Запускать такое задание можно раз в день в одно и то же время, скажем в полночь, когда посещаемость сайта низкая. При таком подходе файлы поставщиков, отправленные в текущие сутки до полуночи, попадут в один пакет. Обычно в случае сбоя пакетные задания выполняются повторно, позволяя удалить или перезаписать данные, созданные во время предыдущего неудачного запуска.

Учитывая природу систем пакетной обработки, при их тестировании помните о следующих общих тестовых сценариях:

- о проверке того, что входные файлы обрабатываются полностью и операция не прерывается на полпути;
- особой обработке поврежденных входных данных, например неожиданных нулевых или очень больших значений и других аномалий;
- маркировке и изоляции неполных записей, которые невозможно преобразовать в нужную структуру;
- реализации в механизме повторных попыток удаления или перезаписи данных после неудачного запуска пакетного задания;
- проверке отсутствия негативного влияния на производительность приложения, если пакетные задания потребляют значительный объем вычислительных ресурсов.

Иногда во время тестирования можете обнаружить, что какие-то стороны начали отправлять данные в новых форматах, что может потребовать обновления пакетного задания. Кроме того, у разных поставщиков может оказаться разное количество единиц товаров в определенных категориях, таких как мужская одежда, спортивная обувь и т. д. То, что количество товаров в одной конкретной категории слишком велико, может быть обусловлено искажением данных (<https://oreil.ly/dTpJ3>) и способно отрицательно сказаться на производительности пакетного задания в зависимости от того, как оно запрограммировано. Поэтому предварительное получение нескольких образцов входных данных от их владельцев поможет в тестировании.

Потоки событий

Под *событием* в буквальном смысле подразумевается действие, а под *потоком* понимается нечто, что течет, или, другими словами, непрерывно по своей природе. Таким образом, *потоки событий* — это системы, в которые постоянно помещаются (публикуются) события, характерные для приложения, и из которых другие системы извлекают (потребляют) эти события, когда у них появляется время для дальнейшей обработки. Например, в приложении электронной коммерции событие создания заказа с подробной информацией публикуется в потоке событий сразу же, как только клиент разместит заказ, а последующие системы извлекают это событие и производят нужные для его исполнения действия (рис. 5.2). С точки зрения потока данных информация о заказах хранится в потоке событий в течение определенного времени, и до его истечения все системы, к которым относятся эти данные, читают их.



Рис. 5.2. Поток событий

Здесь сервис управления заказами Order называется *издателем*, потому что издает (публикует) события, а последующие системы, потребляющие события, называются *подписчиками*. Каждое событие публикуется с определенным названием *темы*, чтобы подписчики могли идентифицировать события, имеющие к ним отношение. Некоторые системы потоков событий, такие как Google Cloud Pub/Sub (<https://cloud.google.com/pubsub/architecture>) и RabbitMQ (<https://www.rabbitmq.com>), удаляют событие из потока после того, как все предполагаемые подписчики извлекли его. В других системах, таких как Apache Kafka (<https://kafka.apache.org>), события удаляются по истечении заданного времени. Такое решение позволяет подписчикам наверстать упущенное после промежуточных сбоев. Потоки событий также обеспечивают надежность, записывая события на диск, подобно базам данных.

Учитывая особенности системы потоковой передачи событий, стоит задать закономерный вопрос: можно ли заменить ее пакетным заданием? Главное отличие пакетной обработки от потоковой заключается в ее ограниченной по времени природе, то есть пакетное задание обрабатывает входные данные в заранее определенные моменты, тогда как потоковая обработка происходит практически в реальном времени. Например, сервис управления заказами в приложении электронной коммерции публикует событие сразу после создания заказа, но не требует подтверждения от последующих систем, то есть данная операция выполняется асинхронно. Благодаря этому размещение заказа в потоке событий позволяет не задерживать обработку заказа на несколько часов и более, как при пакетной обработке, и при этом не выполнять ее синхронно, как запрос веб-сервиса. Такой способ обработки называется *близким к реальному времени*, а не *в реальном времени*, даже притом что события могут быть получены подписчиками в течение нескольких секунд. Эта асинхронная модель хорошо подходит для параллельной обработки и масштабирования. В наши дни она нашла широкое применение в мобильных и веб-приложениях.

Вот некоторые примеры тестов, о которых следует помнить при тестировании системы потоковой передачи событий.

- Структура событий — это соглашение между издателем и подписчиком, поэтому при любом изменении структуры необходимо заново протестировать весь функциональный поток.
- Иногда желательно проверить обратную совместимость, чтобы убедиться в поддержке как старых, так и новых структур событий.
- Иногда события должны обрабатываться в определенной последовательности. Например, отгрузить товар невозможно, пока склад не подтвердит его наличие. Поскольку обработка событий происходит асинхронно, последовательность их обработки необходимо протестировать.
- В случае сбоя подписчик должен иметь возможность повторно обработать события в правильном порядке.
- Если даже после нескольких повторных попыток при обработке какого-то события возникают ошибки, его следует переместить в отдельную очередь, называемую *очередью недоставленных сообщений* (<https://oreil.ly/7Ykw7>), с добавлением информации об ошибке для отладки. Помещение таких событий в очередь недоставленных сообщений тоже необходимо протестировать.
- Что происходит, когда поток событий останавливается? Как издатели и подписчики обрабатывают неудачные попытки обратиться к очереди? Когда и как они повторяют попытки?
- Подписчики могут работать медленнее издателя, что приведет к раздуванию потока. Следовательно, необходимо проверить их способность своевременно извлекать события.

Как видите, каждая из представленных систем хранения и обработки данных играет уникальную роль в более крупной экосистеме и поэтому требует особого внимания при тестировании. Это подводит нас к следующему разделу, где мы рассмотрим особенности тестирования четырех ранее обсуждавшихся систем данных.

Стратегия тестирования данных

В своей книге *Designing Data-Intensive Applications*¹ (<https://oreil.ly/tVDXr>) Мартин Клеппман пишет:

«Было бы неразумно предполагать, что ошибки — редкое явление, и просто надеяться на лучшее. Важно рассматривать широкий круг возможных сбоев — даже очень маловероятных — и искусственно создавать подобные ситуации при тестировании, выясняя, что произойдет».

В этом я с ним полностью согласна, особенно когда дело касается тестирования данных. В тестировании данных 90 % времени уходит на размышления о возможных ошибках, в отличие от функционального тестирования, где мыслительный процесс вращается вокруг вероятных действий пользователя в приложении. Это очевидно вытекает из предыдущего раздела, где мы сосредоточились на тестовых сценариях, вызывающих ошибки.

Учитывая сказанное, типичную стратегию тестирования данных можно представить как состоящую из четырех ветвей (рис. 5.3).

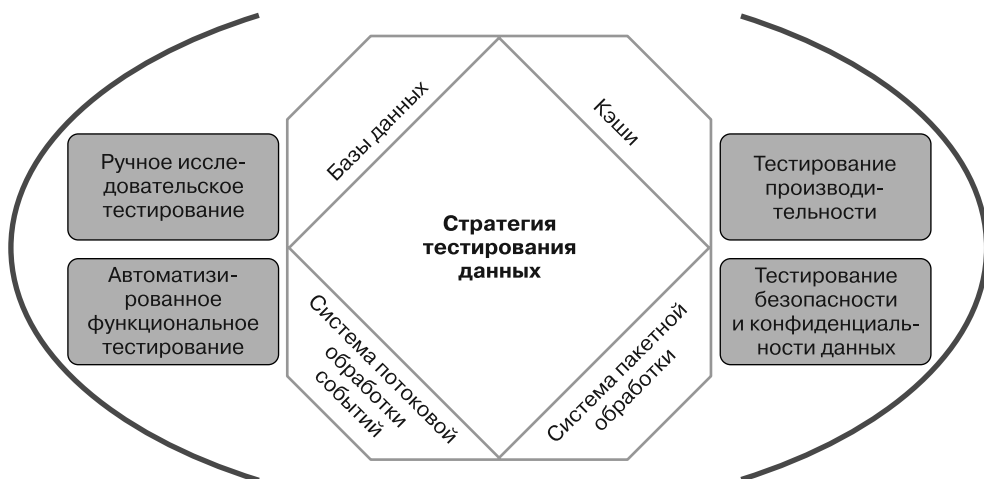


Рис. 5.3. Стратегия тестирования данных

¹ Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — СПб.: Питер, 2019.

Вот эти ветви.

Ручное исследовательское тестирование

Ручное исследовательское тестирование помогает выявить новые тестовые сценарии, вызывающие ошибки, поэтому оно занимает важное место в тестировании данных. В главе 2 вы узнали о методе тестирования данных «Выборка», который с успехом можно применить для тестирования баз данных и систем пакетной обработки. Кроме того, изучение свойств используемых инструментов обработки данных, таких как Apache Kafka, Redis и т. д., поможет выявить конкретные аспекты каждого инструмента, которые стоит проработать вручную.

Возможно, вам потребуется также изучить дополнительные инструменты для ручного исследования. Так, SQL — важный инструмент для исследовательского тестирования реляционных баз данных, и примеры его применения включены в упражнения этой главы.

Автоматизированное функциональное тестирование

Чтобы быстро получить результаты тестирования данных, тесты необходимо автоматизировать и интегрировать в конвейер CI. Тестирование всех четырех систем данных, обсуждаемых в этой главе, рекомендуется начинать с модульного или интеграционного тестирования. Некоторые из нужных для этого инструментов обсуждаются в следующем разделе.

Тестирование производительности

Как мы видели, системы хранения и обработки данных — это важнейшие компоненты любого приложения и их производительность сильно влияет на общую производительность приложения. Поэтому важно провести нагрузочное и стресс-тестирование всех систем хранения и обработки данных, используемых приложением. Такой вид тестирования подробно обсуждается в главе 8.

Безопасность и конфиденциальность

Наличие брешей в системе защиты данных может привести к огромным потерям для клиентов и повлечь за собой серьезные штрафы для бизнеса. Тестирование безопасности — это одна из важнейших сторон тестирования данных, оно будет подробно обсуждаться в главе 7. Кроме того, в каждой конкретной стране существуют свои законы о защите данных, которые предусматривают гарантии конфиденциальности данных. Эти правила и проверка на соответствие им будут обсуждаться в главе 10.

Завершая обсуждение стратегии тестирования данных, вспомним несколько ключевых моментов, отмеченных ранее: при тестировании каждой из этих ветвей нужно учитывать типы и варианты данных, параллелизм, распределенный характер данных и систем, а также вероятность сетевых сбоев. Также имейте в виду, что некоторые тестовые сценарии, связанные с данными, могут не выявить основные ошибки, поскольку они сильно зависят от времени и порядка выполнения действий

(например, тестовые сценарии, связанные с параллелизмом). Обязательно обсудите эти тестовые сценарии на этапе анализа. Далее мы поработаем с некоторыми упражнениями.

Упражнения

В представленных далее упражнениях используется несколько инструментов тестирования баз данных, таких как SQL и JDBC. Мы также рассмотрим Apache Kafka и Zerocode — инструмент для написания автоматизированных тестов с целью проверки очереди сообщений Kafka.



Как упоминалось ранее, большинство тестовых сценариев, связанных с данными, в идеале должно быть реализовано в виде автоматизированных модульных и интеграционных тестов. В этом разделе мы обсудим инструменты, необходимые тестировщикам для ручного исследовательского тестирования или автоматизированного функционального тестирования на макроуровне.

SQL

Без знания SQL невозможно протестировать функции приложения, предполагающие взаимодействие с реляционной базой данных. Вы неизбежно столкнетесь со сценариями, в которых потребуется выполнить запрос к БД и убедиться в целостности данных. В ходе работы с базами данных, содержащими множество таблиц, столбцов и строк, знание приемов фильтрации и получения только тех данных, которые необходимы для тестового сценария, избавит вас от множества проблем. Итак, если вы не знакомы с такими аспектами языка SQL, как сортировка, фильтрация, группировка, вложение, объединение и т. д., то опробуйте это упражнение!

Для его выполнения понадобится реляционная база данных. Если у вас уже есть готовый экземпляр, отлично! Если нет, то выполните действия, описанные в пункте «Предварительные условия», чтобы настроить его.

Предварительные условия

Установите и настройте базу данных PostgreSQL на своем локальном компьютере, загрузив соответствующий пакет с официального сайта (<https://oreil.ly/Qsmp2>). После установки запустите сервер postgres (<https://oreil.ly/gIsxP>), применяя нужные для вашей ОС команды. Например, если вы пользуетесь Mac, то откройте программу Terminal («Терминал») и выполните следующие команды.

1. Загрузите PostgreSQL командой `brew install postgresql`.
2. Запустите сервер Postgres командой `brew services start postgresql`.

3. Запустите клиент командной строки `psql`, выполнив команду `psql postgres`. Клиент `psql` подключится к серверу базы данных и даст возможность выполнять SQL-запросы к нему. Как вариант можно использовать клиент с графическим интерфейсом, например pgAdmin (<https://oreil.ly/rydvr>).



Закончив упражнения, не забудьте остановить сервер базы данных (<https://oreil.ly/4vhii>), например, командой `brew services stop postgresql`.

Рабочий процесс

Как упоминалось ранее, язык SQL используется для выполнения операций с данными (чтения, записи, изменения и удаления) в реляционных базах данных. Язык имеет множество ключевых слов и функций, упрощающих сортировку, фильтрацию и объединение данных из нескольких таблиц. Здесь я покажу запросы, наиболее востребованные при ручном тестировании базы данных.

Создание. Сначала создайте новую таблицу с именем `items`, в которой будут храниться сведения о товаре, такие как артикул, цвет, размер и цена. Для этого выполните следующий запрос из клиента `psql`:

```
postgres=> create table items (item_sku varchar(10), color varchar(3),
size varchar(3), price int);
```

Этот запрос использует ключевые слова SQL `create table`, чтобы указать тип выполняемой операции и присвоить имя таблице. Кроме того, в нем подробно описана структура столбцов: их имена, типы данных (`varchar` и `int` — символьные и целочисленные соответственно) и максимальная длина содержимого. Для трех символьных столбцов длина указывается явно, а для целочисленного столбца подразумевается неявно и равна 4 байтам.



Синтаксис SQL обычно нечувствителен к регистру. Вы можете увидеть ключевые слова, написанные прописными буквами, например `CREATE TABLE`, `VARCHAR`, `INT`. Это не влияет на выполняемую операцию, так что можете использовать любой стиль по своему желанию.

Вставка. Чтобы заполнить таблицу данными — в нашем случае сведениями о товарах, — выполните следующий запрос:

```
postgres=> insert into items values ('ABCD0001', 'Blk', 'S', 200),
('ABCD0002', 'Ye1', 'M', 200);
```

В нем используются три ключевых слова: `insert`, `into` и `values`. Первые два задают тип операции и указывают на таблицу, куда будет сделана вставка (`items`). Значения для вставки указаны в круглых скобках и должны соответствовать порядку столбцов. Аналогично можно вставить столько записей, сколько потребуется. При попытке вставить данные, не укладывающиеся в максимально заданную длину

столбца или не соответствующие указанным типам данных, запрос завершится неудачей. Заполните таблицу дополнительными товарами с разными ценами, разных цвета и размеров, прежде чем переходить к следующим запросам.

Выборка. Наиболее частой операцией тестирования базы данных является *чтение*. Чтобы прочитать данные из таблицы, используйте следующую команду (ваши результаты будут отличаться в зависимости от того, какие данные вы добавили):

```
postgres=> select * from items;
item_sku | color | size | price
-----+-----+-----+-----
ABCD0001 | Red   | S    | 200
ABCD0002 | Blk   | S    | 200
ABCD0003 | Yel   | M    | 200
ABCD0004 | Blk   | S    | 150
ABCD0005 | Yel   | M    | 100
ABCD0005 | Blk   | S    | 120
ABCD0007 | Yel   | M    | 180
(7 rows)
```

Обратите внимание на ключевые слова `select`, `*` и `from` в запросе. Подстановочный знак `*` означает, что необходимо прочитать все строки и столбцы таблицы. Если нужно выбрать конкретные столбцы, то вместо `*` перечислите их имена через запятую.

Фильтрация и группировка. В большинстве случаев таблицы содержат множество строк, а строки имеют n столбцов. Вам может понадобиться отфильтровать данные и оставить только строки, относящиеся к конкретному тестовому примеру. Следующий запрос ограничивает набор возвращаемых результатов:

```
postgres=> select item_sku, size from items limit 3;
item_sku | size
-----+-----
ABCD0001 | S
ABCD0002 | S
ABCD0003 | M
(3 rows)
```

Здесь мы выбираем из таблицы `items` только столбцы `item_sku` и `size`, а ключевое слово `limit` ограничивает возвращаемый набор данных первыми n записями. Также для фильтрации можно использовать ключевое слово `where`, в котором перечисляются критерии фильтрации на основе значений столбцов, например:

```
postgres=> select color from items where size='S';
color
-----
Red
Blk
Blk
Blk
(4 rows)
```

Этот запрос отфильтровал записи, оставив только товары размера S, и вернул только значения в столбце `color`. Как показывают полученные результаты, несколько товаров имеют один и тот же цвет, но из этого наблюдения трудно извлечь смысл. Возможно, будет полезнее просмотреть сводную информацию о количестве товаров размера S каждого цвета. Получить эти сведения можно с помощью ключевого слова `group by`:

```
postgres=> select color, count(*) from items where size='S' group by color;
color | count
-----+-----
Blk   |      3
Red   |      1
(2 rows)
```

Обратите внимание на то, что ключевое слово `group by` можно использовать и без ключевого слова `where`. По сути, оно суммирует несколько строк, опираясь на заданные критерии, и представляет каждую группу как одну строку в результатах запроса. Сгруппированные результаты можно дополнительно отфильтровать с помощью ключевого слова `having`:

```
postgres=> select color, count(*) from items where size='S' group by color
having count(*)>1;
color | count
-----+-----
Blk   |      3
(1 row)
```

Этот запрос фильтрует и оставляет только те из них, в которых количество товаров больше 1. Внимание: ключевое слово `having` можно использовать только вместе с `group by`.

Также обратите внимание на функцию `count(*)` — она подсчитывает количество записей в каждой группе. Далее мы рассмотрим еще несколько функций SQL.

Сортировка. Помимо фильтрации, язык SQL позволяет также сортировать результаты по возрастанию/убыванию на основе значений одного или нескольких столбцов, предлагая ключевое слово `order by`:

```
postgres=> select item_sku, color, size from items order by price asc;
item_sku | color | size
-----+-----+-----
ABCD0005 | Yel   | M
ABCD0005 | Blk   | S
ABCD0004 | Blk   | S
ABCD0007 | Yel   | M
ABCD0001 | Red   | S
ABCD0003 | Yel   | M
ABCD0002 | Blk   | S
(7 rows)
```

Вот пример запроса с сортировкой по нескольким столбцам в разном порядке:

```
postgres=> select * from items order by price asc, size desc;
```

Функции и операторы. В нескольких примерах ранее нам встретилась функция `count()`. Язык SQL предоставляет целый набор функций и операторов для агрегирования, сравнения и других преобразований. Некоторые функции, такие как `sum()`, `avg()`, `min()` и `max()`, можно использовать подобно `count()`. Нетрудно догадаться, что они возвращают сумму, среднее, а также минимальное и максимальное значения соответственно. Для фильтрации могут пригодиться также операторы `and`, `or`, `not` и `null`. Например, попробуйте выполнить следующий запрос, который возвращает товары черного цвета размера S:

```
postgres=> select * from items where size='S' and color='Blk';
```

Выражения и предикаты. В SQL можно использовать также выражения и предикаты. Выражения могут вычислять математические формулы, такие как `price+100`, а предикаты — проверять логические условия, дающие в результате значение `true`, `false` или `unknown`. Например, попробуйте следующий запрос:

```
postgres=> select * from items where price=100+50 and color is not NULL;
```

Язык SQL вычислит выражение, чтобы определить значение цены для фильтрации, а также проверит логическое условие `is not null` для значения цвета в каждой записи, чтобы убедиться, что оно не пустое.

Вложенные запросы. При необходимости запросы можно вкладывать друг в друга. Подзапрос можно поместить в любое место основного запроса, в том числе в предложения `where`, `group by` и т. д. Запрос в следующем примере возвращает общее количество и среднюю цену всех товаров. Обратите внимание на вложенный подзапрос, заключенный в круглые скобки:

```
postgres=> select count(*), (select avg(price) from items) from items;
count |          avg
-----+-----
7 | 164.2857142857142857
(1 row)
```

Соединения. В большинстве случаев данные распределяются по нескольким таблицам, поэтому может потребоваться сопоставить их для проверки тестового сценария. Для выполнения запросов к нескольким таблицам язык SQL предоставляет ключевое слово `join`. Оно позволяет объединить две таблицы на основе их общих атрибутов. Для опробования этого ключевого слова создайте еще одну таблицу с именем `orders` и со столбцами `order_id`, `item_sku` и `quantity` и вставьте несколько строк:

```
postgres=> create table orders (order_id varchar(10), item_sku varchar(10),
quantity int);
postgres=> insert into orders values ('PR123', 'ABCD0001', 1),
('PR124', 'ABCD0001', 3), ('PR125', 'ABCD0001', 2);
```

Теперь можно использовать ключевое слово `inner join`, чтобы объединить таблицы `items` и `orders` на основе `item_sku` — общего столбца для двух таблиц:

```
postgres=> select * from orders o inner join items i on o.item_sku=i.item_sku;
order_id | item_sku | quantity | item_sku | color | size | price
-----+-----+-----+-----+-----+-----+-----
PR124    | ABCD0001 |         3 | ABCD0001 | Red   | S    | 200
PR125    | ABCD0001 |         2 | ABCD0001 | Red   | S    | 200
PR123    | ABCD0001 |         1 | ABCD0001 | Red   | S    | 200
-----+-----+-----+-----+-----+-----+-----
```

Как видите, столбцы из двух таблиц были объединены в одну строку. Но обратите внимание на то, что в соединение включены только элементы `item_sku`, присутствующие в обеих таблицах, — других элементов из таблицы `items` здесь нет. У запроса есть еще несколько особенностей: он использует ключевое слово `on` для описания условия слияния и определяет псевдонимы для таблиц (`o` для `orders` и `i` для `items`), что упрощает применение. Псевдонимы повторно используются в условии соединения.

Помимо внутреннего соединения, часто применяются также *левое соединение*, *правое соединение* и *полное внешнее соединение*. Синтаксис таких запросов остается прежним, меняется только ключевое слово. Левое соединение получает все строки из первой (левой) таблицы в условии соединения и объединяет их с соответствующими строками во второй (правой) таблице. Если строка не имеет совпадений во второй таблице, то соответствующие столбцы заполняются значениями `null`. Правое соединение работает в обратном порядке: принимает все строки из второй таблицы и добавляет столбцы из первой, если в ней имеется соответствующая запись. Полное внешнее соединение возвращает все строки из обеих таблиц, и если совпадающих строк нет, то в объединенных результатах отображаются значения `null`.

Такие запросы на соединение можно дополнить инструкциями фильтрации и сортировки.

Обновление и удаление. Здесь представлены две оставшиеся операции из набора CRUD — запросы на обновление и удаление. Обновить значение столбца можно с помощью ключевых слов `update` и `set`:

```
postgres=> update items set color='BK' where color='Blk';
```

Аналогично, если потребуется удалить некоторые записи, созданные в целях тестирования, используйте ключевое слово `delete`:

```
postgres=> delete from items where price=180;
```



Язык SQL намного богаче, чем можно предположить, прочитав это краткое введение. Как упоминалось ранее, представленные здесь команды используются главным образом в ручном тестировании баз данных. За дополнительной информацией об SQL рекомендую обратиться к руководству *SQL Pocket Guide*¹ (O'Reilly), написанному Элис Жао.

¹ Жао Э. SQL. Pocket guide, 4-е изд. — СПб.: Питер, 2024.

JDBC

Аббревиатура JDBC (<https://oreil.ly/Mg9dZ>) расшифровывается как Java Database Connectivity. Это набор Java API для подключения к реляционным базам данных и выполнения SQL-запросов к ним. JDBC можно применять в любых наборах автоматизированных тестов пользовательского интерфейса или API, упомянутых в предыдущих главах, для непосредственной проверки данных в БД. Для разных баз данных существуют свои драйверы JDBC, которые можно импортировать в проект как зависимость Maven, — например, мы можем использовать драйвер JDBC для PostgreSQL, чтобы подключиться к созданной ранее базе данных PostgreSQL и проверить записи в ней.

В следующем примере с помощью трех простых функций JDBC подключимся к БД и выполним запросы:

```
// Подключиться к базе данных
connection = DriverManager.getConnection("jdbc:postgresql://host/database",
"имя_пользователя", "пароль");

// Выполнить SQL-запрос
Statement statement = connection.createStatement();
ResultSet results = statement.executeQuery(String query);

// Закрыть соединение по окончании
results.close();
statement.close();
```

СОБЛЮДАЙТЕ ПРИНЦИП ПИРАМИДЫ

По причинам, описанным в главе 3, проверки БД следует добавлять как модульные и интеграционные тесты, а не как тесты пользовательского интерфейса или API. Кроме того, идеальный способ создания тестовых данных, необходимых автоматизированным тестам на макроуровне, — задействовать API приложения. Это избавит от необходимости заботиться в тестах о возможных изменениях схемы базы данных, потому что код приложения, лежащий в основе API, автоматически учтет все эти изменения. Сами API должны меняться очень редко, потому что иначе будет затруднена интеграция с клиентами.

Кроме того, иногда может потребоваться выполнить сквозную проверку функциональности, включая нижестоящие системы. Если эти системы унаследованные, то они могут не иметь API, доступного для тестирования. В таких случаях единственный доступный выбор — прямое подключение к базе данных. Так, в примере приложения электронной коммерции для проверки успешности обработки заказа необходимо сначала создать заказ в приложении, а затем напрямую проверить базу данных системы исполнения, если допустить, что это унаследованная система, не имеющая внешнего API. Это упражнение специально предназначено для того, чтобы помочь вам в подобных ситуациях.

Настройка и рабочий процесс

Расширим среду автоматизации Java-Selenium WebDriver, созданную в главе 3, и добавим тест, получающий заказ из таблицы `orders` и проверяющий значения `order_id` и `quantity`. Для этого выполните следующие шаги.

1. Добавьте драйвер PostgreSQL JDBC (<https://oreil.ly/qBf5L>) как зависимость в файл POM.
2. Создайте в пакете `tests` новый файл тестового класса с именем `DataVerificationTest.java`.
3. В примере 5.1 показано объявление класса `DataVerificationTest`, который перед каждым тестом устанавливает соединение с базой данных PostgreSQL и закрывает его по завершении теста. Тест определяет SQL-запрос, который получает записи из БД, и использует утверждения TestNG для проверки возвращаемых данных.

Обратите внимание на JDBC URL, используемый в тесте для подключения к базе данных. В качестве имени хоста берется `localhost`, поскольку БД находится на локальном компьютере. В этом примере база данных имеет имя `postgres`, и вам нужно будет подставить свои имя пользователя и пароль. Вы можете запустить команду `\l` из командной строки клиента `psql`, чтобы просмотреть существующие базы данных, и команду `\dt`, чтобы получить список всех таблиц и их владельцев.

Пример 5.1. Тест, устанавливающий JDBC-соединение с базой данных PostgreSQL

```
package tests;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import static org.testng.Assert.*;
import java.sql.*;

public class DataVerificationTest {

    private static Connection connection;
    private static ResultSet results;
    private static Statement statement;

    @BeforeTest
    public void initiateConnection() throws SQLException {
        connection = DriverManager.getConnection(
            "jdbc:postgresql://localhost/postgres",
            "newuser", null);
    }
}
```

```

public void executeQuery(String query) throws SQLException {
    initiateConnection();
    statement = connection.createStatement();
    results = statement.executeQuery(query);
}

@Test
public void verifyOrderDetails() throws SQLException {
    executeQuery("select * from orders where item_sku='ABCD0006'");
    System.out.println(results);
    while (results.next()){
        assertEquals(results.getString("Quantity"), "1");
        assertEquals(results.getString("order_id"), "PR125");
    }
}

@AfterTest
public void closeConnection() throws SQLException {
    results.close();
    statement.close();
}
}

```

4. Запустить тест можно из командной строки (командой `mvn clean test`) или из IDE. Не забудьте перед запуском теста запустить сервер `postgres`.

Вот так просто. При желании можно вынести методы, связанные с подключением к базе данных, в отдельный класс `utils`, чтобы использовать их повторно в других тестах.

Apache Kafka и Zerocode

Kafka (<https://kafka.apache.org/intro>) — это распределенная платформа с открытым исходным кодом для потоковой обработки данных. Она позволяет нескольким производителям и потребителям (или издателям и подписчикам, если пользоваться терминологией из обсуждения потоковой передачи событий) обмениваться информацией через общий поток. Kafka была разработана командой LinkedIn для решения задач сбора информации из нескольких систем и создания значимых показателей. Она позволила им масштабироваться для обработки триллионов сообщений (<https://oreil.ly/WTQPJ>) и потреблять петабайты данных каждый день.



Довольно интересно, что этот инструмент получил свое название в честь Франца Кафки, известного автора нескольких сюрреалистических произведений, в том числе рассказа *The Metamorphosis*¹, просто потому, что главный инженер, руководивший разработкой, был его поклонником.

¹ *Кафка Ф. Превращение*. — М.: АСТ, 2021.

Познакомимся с этим инструментом поближе, чтобы лучше понять, что это такое и как его тестировать. На рис. 5.4 изображена схема системы на основе Kafka, состоящей из сервера (брокера), производителей и потребителей.

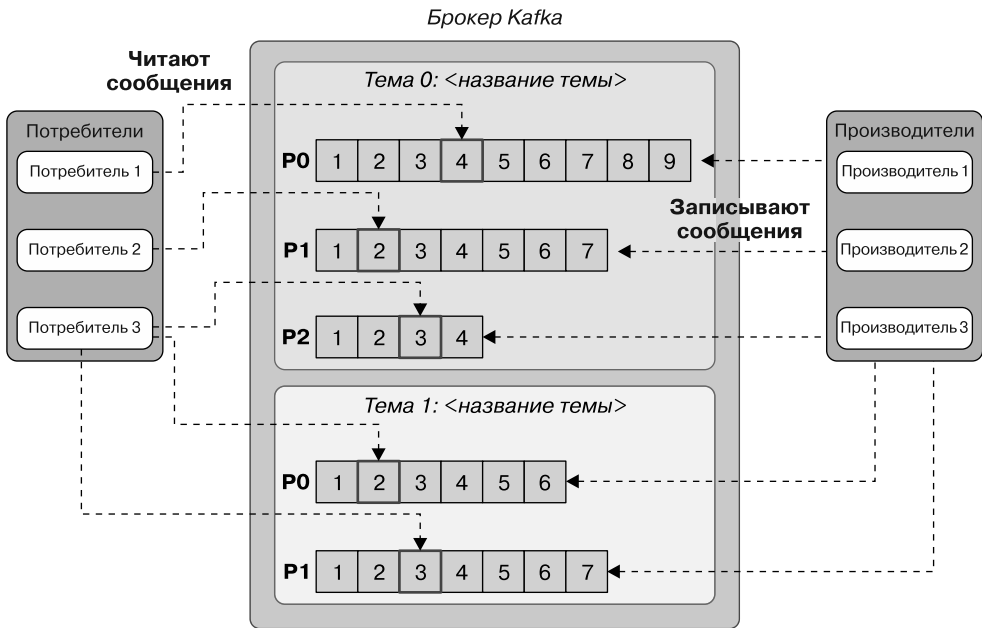


Рис. 5.4. Процесс работы Apache Kafka

Чтобы вы лучше поняли рабочий процесс, перечислю некоторые ключевые термины.

Сообщения

В терминологии Kafka события называют сообщениями. Каждое сообщение несет в себе некоторую информацию, например, сведения о заказе. Сообщения сохраняются непосредственно на диске, что обеспечивает их долговечность.

Темы

Сообщения организованы по названиям тем. Например, в нашем примере сведения о заказе могут быть опубликованы в теме *orders*. В одной теме может иметься множество сообщений, отправленных разными производителями. Это позволяет собирать данные от нескольких производителей и облегчает для потребителей определение актуальных для них сообщений.

Разделы

Сообщения в каждой теме обычно хранятся в нескольких разделах (см. рис. 5.4). Сообщения добавляются в заданный раздел, что дает возможность обрабатывать

их в определенной последовательности, если этого требует рабочий процесс. Для управления распределением сообщений по нужным разделам производители добавляют в сообщения метаданные, называемые *ключами*. Например, если набор транзакций привязан к идентификатору клиента, то он используется как ключ для отправки их в нужный раздел и тем самым обеспечивает возможность обработки транзакций в правильной последовательности.

Разделы — это способ поддержки высокой производительности и масштабируемости, которыми так славится Kafka. Разделы можно реплицировать, чтобы обеспечить избыточность и предотвратить потерю информации из-за непредвиденных сбоев.

Смещение

Каждый потребитель может читать сообщения из множества тем, и нужно как-то отслеживать, какие сообщения он прочитал из каждой темы, чтобы избежать их повторной обработки. Для этой цели потребители используют номер *смещения*, указанный в сообщениях. Смещение — это монотонно увеличивающееся целое число, которое создается платформой Kafka и добавляется в качестве метаданных к каждому сообщению во время его создания. Благодаря этому всякий раз, когда в потребителе происходит сбой, он может возобновить обработку с последнего смещения, которое было успешно обработано. Кроме того, для подключения нового потребителя достаточно просто задать для него более ранний номер смещения, чтобы он мог начать обработку с более ранних сообщений. Эту функцию называют повторным воспроизведением сообщений.

Брокеры

Сервер Kafka, называемый брокером, выступает в роли посредника между производителями и потребителями. Брокер получает сообщения от производителей, добавляет в них смещения и сохраняет на диске, упорядочивая по темам. Аналогичным образом он отвечает на запросы потребителей, извлекая нужные сообщения из нужных разделов.

Схемы

Kafka рассматривает сообщения как простой набор байтов данных, облегчая тем самым взаимодействие между производителями и потребителями — они сами должны договориться о формате и структуре данных, называемых *схемой*. Например, на рис. 5.2 мы видели схему темы `orders`, она состоит из полей `order_id`, `item_sku` и `quantity`.

Kafka поддерживает сообщения в форматах JSON, XML и Apache Avro и многих других. Структура сообщений не может измениться без изменения кода потребителя и производителя. По сути, при наличии разных версий схемы необходимо обеспечить прямую и обратную совместимость. Их можно сравнить

с контрактами запросов и ответов веб-сервисов. Версии схем хранятся в отдельном компоненте, *реестре схем* (<https://oreil.ly/nBVfK>), который помогает проверять совместимость и гарантирует соблюдение контракта между производителем и потребителем при развитии схемы.

Хранение

Kafka хранит сообщения в течение некоторого времени, прежде чем удалить их. По умолчанию они хранятся в течение семи дней или пока размер раздела не достигнет 1 Гбайт. Эти значения можно настроить для каждого сообщения, чтобы удовлетворить требования к сохранности для разных видов сообщений.

Этого вполне достаточно, чтобы начать работу и увидеть, как все работает.

Установка и настройка

Установить Kafka на локальный компьютер можно, выполнив описанные здесь шаги. Поскольку цель — познакомить вас с экосистемой Kafka с точки зрения тестирования, мы не будем углубляться в детали установки и используем контейнеры Docker.

КРАТКОЕ ВВЕДЕНИЕ В DOCKER

Допустим, что вы разрабатываете приложение, которому требуется ряд дополнительных инструментов: база данных PostgreSQL, Kafka, Nginx и т. д. Обычный способ поделиться подробностями установки с новыми членами команды — передать им документ, в котором указаны нужные версии устанавливаемого ПО и их конкретные конфигурации. Это часто становится узким местом, поскольку разные члены команды могут использовать разные версии операционных систем, сталкиваться с проблемами при установке из-за несовместимости с существующими инструментами и т. д. На решение проблем и настройку может уйти несколько дней. Более простой подход — упаковать все необходимое приложению в *контейнер* с помощью Docker (<https://docs.docker.com>) и передать его членам команды. Им останется только установить Docker и загрузить контейнер, который запустит приложение с помощью одной команды.

Docker, по сути, изолирует инфраструктуру и прикладное ПО. Запуск приложения в контейнере эквивалентен наличию внутри основного компьютера изолированного компьютера с программным обеспечением, специфичным для приложения. Ключевое преимущество этого приема заключается в том, что изолированный компьютер можно передавать другим, в отличие от основного. Этот метод особенно полезен при создании сред контроля качества и промышленных сред и дает важное преимущество: вы можете применять одни и те же двоичные файлы приложений повсюду.

Перед установкой Docker важно добавить, что он распространяется бесплатно только для личного использования. Возможно, вам придется соблюдать политику вашей компании в отношении ноутбуков при установке Docker на рабочий ноутбук.

Чтобы установить Kafka с помощью Docker, выполните следующие действия.

1. Установите Docker Desktop, скачав двоичные файлы для конкретной ОС с официального сайта (<https://oreil.ly/hQUgt>). По завершении установки приложения Docker Desktop появится приглашение **Start** (Пуск).
2. После нажатия кнопки **Start** (Пуск) вам будет предложено запустить команду `docker run -d -p 80:80 docker/getting-started`. Попробуйте выполнить ее в терминале, чтобы убедиться, что Docker доступен из командной строки. Эта команда загрузит образец контейнера `hello-world` и откроет доступ к нему через порт 80 хост-компьютера.
3. После запуска контейнер `hello-world` появится в приложении Docker Desktop. Остановите контейнер, как только он запустится, щелкнув на кнопке **Stop** (Стоп) рядом с ним.
4. Вскоре я расскажу о Zerocode, а пока просто клонируйте репозиторий Zerocode Docker Factory (<https://oreil.ly/Wg08y>) командой `git clone`. (Инструкции по работе с Git вы найдете в главе 4.) Этот репозиторий содержит все необходимые конфигурационные файлы и зависимости для Kafka, он позволит вам писать автоматизированные тесты с помощью инструмента Zerocode.
5. В завершение установки Kafka выполните в терминале команду `cd`, чтобы перейти в папку `zerocode-docker-factory/compose`, а затем команду:

```
$ docker-compose -f kafka-schema-registry.yml up -d
```

Как только вы увидите зеленую надпись `done` (готово) в выводе команды, запустите команду `docker ps`, которая должна показать, что в данный момент выполняются несколько контейнеров.

Теперь, когда Kafka и все ее зависимости успешно работают на вашем компьютере, можно с помощью Zerocode писать автоматизированные тесты, создающие и потребляющие сообщения.

Рабочий процесс

Zerocode — это инструмент с открытым исходным кодом, который позволяет писать автоматизированные тесты в декларативном стиле для REST API, SOAP API и систем Kafka. Тестовые сценарии можно создавать в виде файлов JSON или YAML и подключать их как обычные тесты JUnit. Тесты могут обращаться к API и проверять сообщения, создаваемые им в Kafka, и наоборот. Также в тестах можно создавать новые сообщения и проверять их структуру при потреблении. Основное преимущество этого инструмента — уровень абстракции, который скрывает вызовы Kafka API, необходимые для выполнения этих операций, а также код сериализации/десериализации для чтения различных типов ответов.

Воспользуемся Zerocode и отправим JSON-сообщение с информацией о заказе, содержащее значения `order_id`, `item_sku` и `quantity`, локальному брокеру Kafka, который вы только что создали, написав декларативные тестовые сценарии. Сообщение будет опубликовано в теме `orders`. Затем мы извлечем сообщение, как это сделала бы система исполнения заказов, и проверим содержащуюся в нем информацию, снова написав декларативные тестовые сценарии. Благодаря этому вы узнаете, как выглядят сообщения и какие детали нужно проверить.

Вот пошаговое руководство по созданию тестов с использованием Zerocode.

1. Создайте новый проект Maven в IntelliJ, например `KafkaTesting`, с помощью Java 1.8 JDK. Если на этом этапе у вас возникнут затруднения, обратитесь к главе 3.
2. Добавьте JUnit 4 и библиотеку `ZeroCode-tdd` в файл `pom.xml`, как показано в примере 5.2.

Пример 5.2. Файл `pom.xml` для тестирования с помощью Zerocode

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>KafkaTesting</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.jsmart</groupId>
      <artifactId>zerocode-tdd</artifactId>
      <version>1.3.28</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

3. Создайте новую папку с именем `src/main/resources`.
4. В эту папку добавьте три файла свойств: `broker.properties`, `producer.properties` и `consumer.properties`, содержимое которых показано в примере 5.3. Эти файлы содержат подробную информацию о каждом контейнере брокера, производителя и потребителя, работающих на вашем локальном компьютере.

Пример 5.3. Файлы свойств Kafka для тестирования

```
// broker.properties

kafka.bootstrap.servers=localhost:9092
kafka.producer.properties=kafka_servers/producer.properties
kafka.consumer.properties=kafka_servers/consumer.properties
consumer.commitSync = true
consumer.commitAsync = false
consumer.fileDumpTo= target/temp/demo.txt
consumer.showRecordsConsumed=false
consumer.maxNoOfRetryPollsOrTimeouts = 5
consumer.pollingTime = 1000
producer.key1=value1-testv ycvb

// producer.properties

client.id=zerocode-producer
key.serializer=org.apache.kafka.common.serialization.StringSerializer
value.serializer=org.apache.kafka.common.serialization.StringSerializer

// consumer.properties

group.id=consumerGroup14
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
max.poll.records=2
enable.auto.commit=false
auto.offset.reset=earliest
```

5. Теперь создайте в `src/main/resources` еще одну папку с именем `test_cases`.
6. Добавьте в папку `test_cases` новый JSON-файл `orderMessages.json`. Сюда вы будете записывать свои тесты в формате JSON.
7. Сначала напишем тест, создающий сообщение с информацией о заказе и проверяющий метаданные, полученные в ответ от брокера. Обычно в ответе возвращаются статус (как и в случае с API), номер раздела и название темы. Файл `orderMessages.json` (то есть тестовый сценарий производителя) показан в примере 5.4.

Пример 5.4. Тестовый сценарий для тестирования производителя в Zerocode

```
{
  "scenarioName": "Produce an order details JSON message for the orders topic",
  "steps": [
    {
```



```

    "name": "produce order messages",
    "url": "kafka-topic:orders",
    "operation": "produce",
    "request": {
        "recordType" : "JSON",
        "records": [
            {
                "value": {
                    "order_id" : "PR125",
                    "item_sku" : "ABCD0006",
                    "quantity" : "1"
                }
            }
        ]
    },
    "verify": {
        "status": "Ok",
        "recordMetadata": {
            "topicPartition": {
                "partition": 0,
                "topic": "orders"
            }
        }
    }
}
]
}

```

8. Далее свяжем тестовый сценарий JSON с тестом JUnit. Для этого создайте в `src/test/java` новый файл `ProducerTest.java` и добавьте в него код из примера 5.5.

Пример 5.5. Класс `ProduceTest`

```
// ProducerTest.java
```

```

import org.jsmart.zerocode.core.domain.JsonTestCase;
import org.jsmart.zerocode.core.domain.TargetEnv;
import org.jsmart.zerocode.core.runner.ZeroCodeUnitRunner;
import org.junit.Test;
import org.junit.runner.RunWith;

@TargetEnv("kafka_servers/broker.properties")
@RunWith(ZeroCodeUnitRunner.class)
public class ProducerTest {

    @Test
    @JsonTestCase("testCases/orderMessages.json")
    public void verifySuccessfulCreationOfOrderDetailsMessageInBroker()
        throws Exception {

    }
}

```

Атрибут `@TargetEnv` сообщает тесту, где искать конфигурацию брокера, атрибут `@RunWith` связывает Zerocode с JUnit, а атрибут `@JsonTestCase` ссылается на файл JSON с тестовым сценарием, который нужно запустить в процессе тестирования.

9. Тест можно запустить из IntelliJ IDE, щелкнув правой кнопкой мыши на зеленой кнопке рядом с атрибутом `@Test`. Как только он выполнится, вы сможете найти сообщения, созданные в локальном экземпляре Kafka, выполнив следующие команды в терминале:

```
// чтобы попасть внутрь контейнера
$ docker exec -it compose_kafka_1 bash

// чтобы увидеть запись такой, какой ее увидит потребитель
$ kafka-console-consumer --bootstrap-server kafka:29092
  --topic orders --from-beginning
```

Поздравляю, вы написали свой первый тест Kafka!

10. Теперь напишем тест, играющий роль потребителя, чтобы проверить содержимое сообщения. Для этого добавьте код JSON, показанный в примере 5.6, в массив `steps` в файле `orderMessages.json`.

Пример 5.6. Тест, потребляющий сообщение с использованием Zerocode

```
{
  "name": "consume order messages",
  "url": "kafka-topic:orders",
  "operation": "consume",
  "request": {
    "consumerLocalConfigs": {
      "recordType": "JSON",
      "commitSync": true,
      "showRecordsConsumed": true,
      "maxNoOfRetryPollsOrTimeouts": 3
    }
  },
  "assertions": {
    "size": 1,
    "records": [
      {
        "value": {
          "order_id": "PR125",
          "item_sku": "ABCD0006",
          "quantity": "1"
        }
      }
    ]
  }
}
```

Потребительский тест проверяет количество полученных сообщений, их темы и содержимое. Zerocode позволяет в том же декларативном стиле добавлять проверки других частей содержимого сообщений, таких как смещение, раздел, ключи/значения и т. д.

Дополнительную информацию об этом способе тестирования Kafka можно найти в официальной документации Zerocode (<https://github.com/authorjapps/zerocode>).

Дополнительные инструменты тестирования

Цель этого раздела — пролить свет на некоторые другие инструменты, помимо тех, что обсуждались ранее, которые обычно используются при тестировании данных и помогают получить более широкое представление о нем.

Тестовые контейнеры

Предварительными условиями для некоторых предыдущих упражнений были настройка реальной базы данных PostgreSQL на локальном компьютере и создание соответствующих таблиц, чтобы тесты могли подключаться к базе данных и проверять данные. Если в вашей тестовой среде есть доступ к базе данных приложения, то тесты могли бы подключиться к ней. Альтернативное решение — использование инструмента Testcontainers (<https://oreil.ly/v7TqQ>), который предоставляет контейнерные одноразовые экземпляры базы данных. Они могут пригодиться разработчикам, желающим запускать модульные и интеграционные тесты на своих локальных компьютерах без предварительной настройки соответствующей базы данных. Даже если бы у них была база данных, высока вероятность того, что она будет засорена в ходе активной разработки функций. Тестовые контейнеры устраняют этот недостаток, каждый раз создавая новый экземпляр базы данных в одном и том же стабильном состоянии, что жизненно важно для успешного тестирования.

Чтобы код приложения мог использовать базу данных Testcontainers, необходимо немного изменить JDBC URL. Также, если это необходимо, для создания экземпляров базы данных можно задействовать API. Например, чтобы вызвать контейнерную базу данных PostgreSQL перед запуском теста, в метод настройки теста можно добавить следующие строки:

```
PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>(A_sample_image);  
postgres.start();
```

При необходимости можно продолжить использовать объект-контейнер в других тестах. Testcontainers предоставляет различные экземпляры баз данных, включая MySQL, PostgreSQL, Cassandra, MongoDB и т. д. Этот подход увеличивает

преимущества работы с JUnit и позволяет запускать базы данных перед тестированием с помощью сценария инициализации, функции или файла.



Testcontainers позволяет запускать множество других типов контейнеров, помимо баз данных (контейнеры Kafka и RabbitMQ, контейнеры с браузерами и т. д.), с помощью всего одной строки кода. Он также предоставляет общую структуру для импорта любых других пользовательских контейнеров, которые могут вам понадобиться для запуска тестов. Полную информацию смотрите в документации.

ТЕСТИРОВАНИЕ ПЕРЕНОСИМОСТИ

Переносимость — это способность приложения отключать свои внутренние компоненты без особой доработки. Иногда переносимость баз данных становится важным требованием при разработке продукта. Представьте, что команда разрабатывает систему управления заказами (order management system, OMS). Ожидается, что OMS как продукт будет подключаться к любой платформе электронной коммерции, используемой в настоящее время бизнесом и внутренними унаследованными системами. В таких случаях переносимость баз данных (другими словами, возможность работать с различными типами баз данных, такими как Oracle или PostgreSQL) может стать ключевым аргументом в пользу продукта OMS, поскольку это означает, что ИТ-команде предприятия не придется изучать новые инструменты. С этой целью команда разработчиков OMS могла бы применять Testcontainers для проверки взаимодействия приложений с различными базами данных в рамках модульного и интеграционного тестирования.

Deequ

Ранее в этой главе мы говорили о производителях, отправляющих свои обновленные каталоги в наше приложение электронной коммерции в виде файлов, а также о пакетных заданиях, преобразующих эти файлы в записи базы данных. Как показывает мой опыт работы над проектами интернет-торговли, даже ведущие ретейлеры в США и Европе продолжают эксплуатировать устаревшие системы, написанные на COBOL, и мейнфреймы. Они каждый день отправляют миллионы записей с данными из своих каталогов в виде файлов для обновления доступности товаров, а по ночам запускаются пакетные задания, вставляющие эти сведения в базу данных приложения. Записи в файлах часто содержат устаревшие или неправильные данные, пустые значения, отсутствующие ключи и множество других огрехов. Это действительно кошмарный сценарий, поскольку, попадая в базу данных, такие данные могут нарушить работу приложения. Deequ (<https://github.com/aws-labs/deequ>) — это инструмент модульного тестирования с открытым исходным кодом, который может помочь в подобных ситуациях.

Deequ был создан в Amazon (<https://oreil.ly/RSxAD>), где продолжает применяться и по сей день для тестирования качества данных, создаваемых внутренними системами компании. Инструмент построен на основе Apache Spark — крупномасштабной

платформы распределенной обработки данных. Как упоминалось ранее, Spark, помимо прочего, можно использовать для пакетной обработки крупномасштабных данных. С помощью библиотеки Deequ можно выполнять модульное тестирование данных до и после пакетной обработки. Например, с помощью Deequ можно добавить модульные тесты, проверяющие ожидаемые типы данных, отсутствие пустых значений, наличие только определенных разрешенных значений и т. д. В нашем приложении электронной коммерции такие модульные тесты можно использовать для проверки данных перед их загрузкой в Spark для пакетной обработки. В ходе тестирования все записи в файле, не соответствующие требованиям, будут помещены в карантин для дальнейшего анализа. Можно также написать набор модульных тестов для проверки преобразованных данных после пакетной обработки для выявления ошибок в самом задании пакетной обработки.

Вот как выглядит пример теста, написанный с помощью Deequ:

```
val verificationResult = VerificationSuite()  
  .onData(data)  
  .addCheck(  
    Check(CheckLevel.Error, "unit testing vendor files")  
      .hasSize(_ > 100000)    // ожидается более миллиона записей  
      .isComplete("item_sku") // это поле не должно иметь пустых значений  
      .isUnique("item_sku")   // эти значения не должны повторяться  
      // следующее поле может иметь только одно  
      // из значений: "S", "M", "L", "XL"  
      .isContainedIn("size", Array("S", "M", "L", "XL"))  
      .isNonNegative("price") // не должно содержать  
                              // отрицательных значений  
  )  
  .run()
```

В процессе тестирования Deequ генерирует различные показатели качества для всех записей. Например, результаты могут показать: 90 % значений в поле `price` приемлемые, а 10 % — нет, что говорит о необходимости их исправления. Инструмент предлагает и другие возможности, такие как обнаружение аномалий в показателях качества данных, автоматические предложения по проверке и т. д.



TensorFlow Data Validation (<https://oreil.ly/6c61n>) и Great Expectations (<https://oreil.ly/dS2D5>) — два других инструмента проверки данных, похожие на Deequ.

На этом обзор тестирования данных завершается. Мы рассмотрели множество вопросов: вы познакомились с разными типами систем хранения и обработки данных, увидели новые тестовые сценарии, которые они добавляют в арсенал функционального тестирования, прошли краткий курс SQL и проработали несколько практических упражнений с применением ряда полезных инструментов, которые помогут вам при разработке собственных проектов. Навыки тестирования данных — крайне необходимый навык в отрасли.

Ключевые выводы

- В наши дни данные составляют основу любого онлайн-приложения, а вокруг них вращаются функциональные возможности, дизайн пользовательского интерфейса, брендинг и маркетинг. Нарушение целостности данных наносит ущерб репутации компаний и порождает недовольство у клиентов. По этим причинам целостность данных невозможно переоценить, а навыки их тестирования считаются одними из основных.
- К навыкам тестирования данных относится знание различных систем их хранения и обработки (включая их уникальные свойства), конкретных тестовых сценариев, а также методов и инструментов тестирования, необходимых для автоматизированного и ручного исследовательского тестирования.
- В этой главе обсуждались четыре широко используемые системы хранения и обработки данных: базы данных, кэши, системы пакетной обработки и системы обработки потоков событий. Мы коснулись их отличительных свойств и рассмотрели тестовые сценарии для каждой из них.
- Типичная стратегия тестирования данных должна включать ручное исследовательское тестирование, автоматизированное функциональное тестирование, тестирование производительности и тестирование безопасности и конфиденциальности данных. Как правило, при тестировании данных вы должны использовать тестовые сценарии, проверяющие сами данные и их варианты, а также распределенную и параллельную обработку и реакцию на сетевые сбои.
- При тестировании данных необходимо применять подход, фокусирующийся на поиске ошибок, поскольку 90 % тестирования данных связано с ошибочными тестовыми примерами. В этом отличие от функционального тестирования, где мыслительный процесс вращается вокруг действий пользователя.

Визуальное тестирование

Визуальное качество придает дополнительную ценность бренду!

Визуальное качество приложения формирует первые впечатления клиентов. Увидев красивое и удобное приложение, они, скорее всего, продолжат изучение его возможностей. Представьте, что клиент должен совершить онлайн-платеж, а кнопка **Continue** (Продолжить) выглядит так, как на рис. 6.1. Как вы думаете, хватит ли у него доверия, чтобы двигаться дальше? Я очень в этом сомневаюсь. Я бы выбрала веб-сайт конкурента, чтобы сделать то, что мне нужно, вместо того чтобы рисковать потерять деньги!

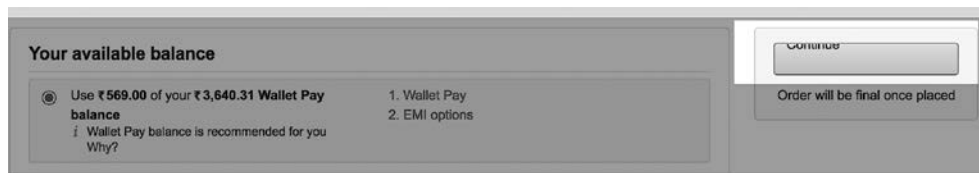


Рис. 6.1. Пример пользовательского интерфейса с обрезанной кнопкой

Компании тратят много средств на стратегии привлечения клиентов с помощью рекламных кампаний, бесплатных акций и многого другого, поэтому команды разработчиков ПО упускают из виду визуальное качество. Но это все равно что построить роскошный дом и забыть его покрасить. Визуальное качество приложения — важнейший фактор, который приближает бизнес к клиенту и помогает завоевать его расположение, а расположение клиента напрямую увеличивает ценность бренда. Визуальное тестирование — это проверка визуального качества приложения с использованием как ручных, так и автоматизированных методов тестирования.



Обратите внимание на то, что тестирование пользовательского опыта (User eXperience, UX) не относится к визуальному тестированию, а подпадает под тестирование удобства использования и обсуждается в главе 10.

Визуальное тестирование предполагает подтверждение соответствия приложения ожидаемому внешнему виду с точки зрения размера и цвета каждого элемента, их относительного расположения и аналогичных визуальных атрибутов на разных устройствах и в разных браузерах. В этой главе будут представлены компактный обзор визуального тестирования с акцентом на обязательных сценариях использования для конкретного проекта/бизнеса и практические упражнения с применением инструментов Cypress и BackstopJS. Здесь же вы найдете обзор нового инструмента автоматизации визуального тестирования на базе искусственного интеллекта Applitools Eyes. Кроме того, мы рассмотрим ландшафт тестирования внешнего интерфейса и увидим, как различные его типы наряду с визуальным тестированием способствуют проверке визуального качества приложения.

Введение

Начнем с введения в визуальное тестирование и различные его методологии, а затем проведем анализ и определим, когда компромиссы между затратами и выгодами становятся критическими для проекта.

Введение в визуальное тестирование

Для проверки визуального качества приложения многие команды разработчиков ПО даже сегодня часто полагаются на визуальный осмотр и автоматизированное тестирование на основе пользовательского интерфейса. Для некоторых приложений этого может быть достаточно, но важно понимать компромиссы, связанные с этим подходом.

Прежде всего согласимся с тем, что человеческий глаз не способен заметить изменения на уровне пикселей, поэтому простое визуальное наблюдение позволяет добиться лишь определенного уровня точности. Например, довольно легко пропустить такие детали, как скругленные края кнопок или логотип, сдвинутый вверх или вниз на несколько пикселей. Исследование (<https://oreil.ly/BaE1m>), проведенное в 2012 году, показало, что наблюдатели не замечают изменений в пятой части изображений. Этот феномен, называемый *слепотой к изменениям*, не имеет ничего общего с дефектами зрения — он чисто психологический. Итак, представьте, что небольшие изменения в приложении могут легко остаться незамеченными при ручном тестировании. Кроме того, не будем забывать о времени и усилиях, которые требуются для ручного визуального тестирования приложения на множестве браузеров, устройств и комбинаций разрешений экрана. Очевидно, что здесь нужна некоторая автоматизация.

Но одних только автоматизированных функциональных тестов на основе пользовательского интерфейса (хотя они частично способствуют проверке визуального качества) может быть недостаточно, потому что они не оценивают внешний вид

элементов, а просто идентифицируют элемент по его локатору, например идентификатору или пути XPath, и проверяют, работает ли он так, как ожидалось. Например, тесты пользовательского интерфейса, изображенного на рис. 6.1, выполнены бы успешно, потому что кнопка **Continue** (Продолжить) обнаруживается тестами в соответствии с ее локатором и ожидаемой меткой и при нажатии успешно перенаправляет пользователя на следующую страницу. Мы здесь не можем винить тест, потому что он вполне соответствует своему предназначению — проверке пользовательского сценария. Еще одно замечание, касающееся применения функциональных тестов на основе пользовательского интерфейса для визуального тестирования: вы не можете добавлять тесты для проверки присутствия каждого элемента на каждой странице приложения, потому что это значительно замедлит их выполнение и потребует больших усилий по обслуживанию.

К счастью, для преодоления этих проблем существуют вполне зрелые инструменты автоматизированного визуального тестирования, подобные инструментам автоматизированного функционального тестирования. Они существуют уже довольно давно, базируются на различных методологиях визуального тестирования и со временем стали более стабильными и простыми в применении. Перечислю некоторые методы работы существующих инструментов для визуального тестирования:

- написание кода для проверки CSS-аспектов элементов (например, тест, проверяющий соответствие условию `border-width=10px`);
- анализ статического кода CSS для выявления несовместимости между браузером и элементами пользовательского интерфейса;
- применение искусственного интеллекта для распознавания изменений на странице, как это делает человеческий глаз;
- создание скриншота страницы и сравнение его попиксельно с ожидаемым базовым скриншотом.

Последний из этих методов наиболее часто используется в регрессионном визуальном тестировании. По этой причине его иногда называют *тестированием скриншотов*. Такого рода визуальное тестирование выполняют некоторые инструменты с открытым исходным кодом, например PhantomJS и BackstopJS. Существуют также коммерческие инструменты, такие как AppliTools Eyes и Functionize, работающие на базе искусственного интеллекта. Эти инструменты можно задействовать для автоматизации визуального тестирования после ручного сравнения приложения с эталонным дизайном и с их помощью выявлять визуальные ошибки, подобно тому как с помощью автоматизированного функционального тестирования обнаруживаются функциональные ошибки. В ходе итеративной разработки визуальные тесты будут постоянно давать вам обратную связь о визуальном качестве приложения.

Важная особенность автоматизированных визуальных тестов, которую следует отметить: они могут стать нестабильными в итеративном процессе разработки, если не добавить их на нужном этапе. Представьте, что ваша команда решила реализовать функцию входа в систему как часть двух пользовательских историй,

одна из которых определяет базовую функциональность, а другая совершенствует функциональность и внешний вид. Добавление функциональных тестов на основе пользовательского интерфейса в рамках этих двух пользовательских историй имеет определенный смысл, но добавление визуальных тестов в части первой истории не обязательно увеличит ее ценность. Поэтому, планируя итерации, включайте визуальное тестирование только в подходящие пользовательские истории.

Критически важные для проекта/бизнеса варианты применения

Мы обсудили, почему важно добавлять автоматизированные визуальные тесты, но это не может быть обязательным для всех приложений. Важным фактором является стоимость. Стоимость тестирования накапливается со временем. В любом проекте в первую очередь приходится платить за разработку и поддержку функциональных тестов на основе пользовательского интерфейса, абсолютно обязательных для всех приложений. Кроме того, у нас есть затраты на разработку и поддержку визуальных тестов даже при том, что два типа тестов можно объединить в один набор. Поэтому, принимая решение об обязательности или полезности автоматизированного визуального тестирования, важно учитывать характер приложения. Например, при разработке внутреннего приложения, которым будут пользоваться лишь несколько администраторов, вероятно, не стоит тратить время и силы на создание автоматизированных визуальных тестов — ручного визуального тестирования будет вполне достаточно. Однако в некоторых из перечисленных далее случаев автоматизированное визуальное тестирование может принести выгоду, достаточную для того, чтобы оправдать затраты.

- Для приложения «бизнес — клиент» (Business-to-Customer, B2C) качество изображения является важнейшим атрибутом общего качества. Поэтому вам нужна постоянная обратная связь по этому аспекту приложения во время разработки. Например, при создании глобального веб-сайта электронной коммерции с большим количеством компонентов на каждой странице постоянная обратная связь по визуальному качеству необходима точно так же, как необходима постоянная обратная связь по функциональности, которую обеспечивают тесты на основе пользовательского интерфейса. В подобных случаях, если только вы не разрабатываете прототип для оценки потребностей рынка и не планируете позже улучшать его дизайн, автоматизированные визуальные тесты помогут создать стабильное приложение.
- Когда необходимо поддерживать приложение, работающее в нескольких браузерах, на различных устройствах и с разными разрешениями экрана, автоматизированные визуальные тесты помогут справиться с огромной нагрузкой регрессионного тестирования.

На рис. 6.2 показана статистика использования веб-сервисов по устройствам, браузерам, производителям, ОС и разрешениям экрана по состоянию на

март 2022 года, согласно данным gs.statcounter.com (<https://gs.statcounter.com>). Как видите, пользователей мобильных устройств больше, чем пользователей настольных компьютеров. Chrome занимает наибольшую долю рынка браузеров, за ним следует Safari. Среди операционных систем важными игроками являются Android, Windows и iOS. Тестирование визуального качества приложения во всех этих комбинациях может легко превратиться в круглосуточную работу, а автоматизированное визуальное тестирование значительно уменьшит усилия вашей команды.

Доли рынка смартфонов, настольных и планшетных компьютеров		Доли рынка браузеров		Доли рынка производителей устройств	
Смартфоны	56,45 %	Chrome	64,53 %	Samsung	28,22 %
Настольные компьютеры	41,15 %	Safari	18,84 %	Apple	27,57 %
Планшетные компьютеры	2,40 %	Edge	4,05 %	Xiaomi	12,24 %
Доли рынка ОС		Firefox	3,40 %	Huawei	6,53 %
		Samsung Internet и Opera	~ 5 %	Oppo	5,25 %
		Доли рынка разрешений экрана			
		1920 × 1080	9,27 %	360 × 800	5,35 %
		1366 × 768	7,32 %	1536 × 864	4,05 %
Android	41,56 %				
Windows	31,15 %				
iOS	16,85 %				
OS X	6,30 %				

Рис. 6.2. Статистика использования веб-сервисов по устройствам, браузерам, ОС и разрешениям экрана, по данным statcounter

- Обычно предприятия, владеющие наборами приложений, имеют централизованную команду, которая разрабатывает компоненты пользовательского интерфейса, образующие *единую систему дизайна*, и несколько команд, повторно применяющих эти компоненты. Например, компоненты пользовательского интерфейса, такие как панели навигации в заголовке с такими элементами, как «Часто задаваемые вопросы», «Свяжитесь с нами» и «Поделиться в социальных сетях», разрабатываются одной командой и повторно используются во всем наборе приложений. В таких случаях визуальное тестирование на уровне компонентов становится необходимостью, потому что любой недочет в стандартных компонентах будет распространяться на весь пакет.

- Иногда приложение полностью перестраивается для улучшения масштабируемости и других аспектов качества, но ожидается, что пользовательский опыт останется таким, какой он есть, потому что клиенты уже обладают им. Написание визуальных тестов может послужить подстраховкой для команд, работающих над таким приложением.
- Точно так же визуальные тесты пригодятся при проведении значительного рефакторинга существующего приложения. Например, повышение эффективности внешнего интерфейса может потребовать значительной реорганизации компонентов пользовательского интерфейса. В такой ситуации автоматизированные визуальные тесты дадут команде большую уверенность.
- Когда приложение расширяется для того, чтобы охватить аудиторию в разных странах, в него включаются такие функции локализации, как особый внешний вид для каждого региона и текст на родном языке. Эти изменения могут повлиять на макет страницы. А когда нужно протестировать несколько версий, автоматизация визуальных тестов может оказаться большим подспорьем.

Таким образом, принимая решение о необходимости реализации автоматизированного визуального тестирования для приложения, учитывайте такие факторы, как влияние на клиента, виды работ, которые требуется выполнить, уверенность команды и объем усилий на ручное тестирование. По мере возможности старайтесь сбалансировать свою стратегию тестирования пользовательского интерфейса и разрабатывайте минимальный набор визуальных тестов только для наиболее важных путей. В следующем разделе представлены различные типы тестов, которые может включать такая стратегия.

Стратегия тестирования фронтенда

Автоматизированное визуальное тестирование может дать преимущества, только если оно сбалансировано с другими типами тестов фронтенда. Знание того, как выглядят различные элементы мозаики тестирования фронтенда, поможет собрать их вместе в соответствии с требованиями вашего приложения. Вы также можете заметить, что некоторые другие типы тестов фронтенда сами по себе способствуют визуальному тестированию. Это следует учитывать при планировании стратегии визуального тестирования приложения.

Кроме того, крайне важно понимать, где и как применяются автоматизированные визуальные тесты, чтобы команды не предлагали их в качестве решения посторонних проблем. Например, очевидно, что вам не нужно добавлять визуальные тесты для всех типов сообщений об ошибках, появляющихся на странице, — это работа модульных тестов пользовательского интерфейса. Поэтому давайте уменьшим масштаб и рассмотрим стратегию тестирования фронтенда в целом.

Код фронтенда веб-приложения состоит из трех основных частей: HTML-кода, определяющего базовую структуру страницы, CSS-кода, определяющего стили элементов на странице, и сценариев, задающих поведение этих элементов. Еще один важный компонент — браузер, отображающий этот код. Большинство новых браузеров придерживаются стандартов отображения элементов. В результате среды разработки фронтенда могут обеспечить встроенную поддержку различных браузеров. Это означает, что элементы или функции, созданные с помощью этих платформ, гарантированно корректно отображаются в основных браузерах, но вам, возможно, придется проверить наличие проблем с совместимостью браузеров при использовании функций, которые платформы не тестируют в старых и новых браузерах.

Для проверки различных частей кода фронтенда мы можем применять различные типы тестов на микро- и макроуровне. Обычно разработчики и тестировщики совместно владеют этими тестами. На рис. 6.3 показано, как можно задействовать различные тесты фронтенда на микро- и макроуровне на протяжении всего процесса разработки, чтобы быстрее получить обратную связь, — другими словами, рисунок проливает свет на реализацию раннего тестирования фронтенда. Мы обсудили некоторых из этих типов тестов в главе 3, а в этом разделе рассмотрим их с точки зрения кода фронтенда, чтобы понять, как они могут помочь выполнять визуальное тестирование.

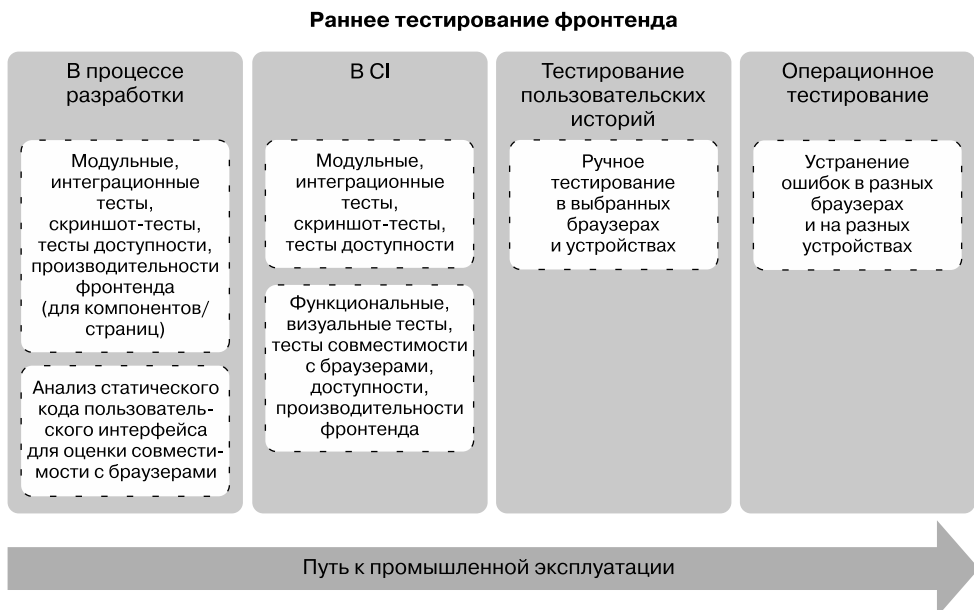


Рис. 6.3. Реализация раннего тестирования фронтенда

Модульные тесты

Модульные тесты фронтенда пишутся на уровне компонентов и проверяют их поведение в различных состояниях. Они также частично помогают выполнить визуальное тестирование. Например, модульные тесты могут проверять вывод приветственного сообщения в заголовке страницы или состояние (активна/неактивна) кнопки отправки. Обычно разработчики пишут эти тесты в начале разработки, используя такие инструменты, как Jest и React Testing Library. Они находятся внутри базы кода и обеспечивают быструю обратную связь на этапе разработки.

В примере 6.1 показан пример модульного теста для проверки приветственного сообщения. Как видите, он извлекает заголовок `h1` и проверяет содержащийся в нем текст. Проверяя элемент `h1`, мы вносим вклад в визуальное тестирование.

Пример 6.1. Пример модульного теста с использованием Jest

```
describe("Component Unit Testing", () => {
  it('displays greeting message as a default value', () => {
    expect(enzymeWrapper.find("h1").text()).toContain("Good Morning!")
  })
})
```

Интеграционные/компонентные тесты

Эти тесты написаны для проверки функциональности компонентов и интеграции между ними, например для контроля поведения формы входа, как показано в примере 6.2. Здесь проверяется функциональность всей формы, а не только одного компонента, как при модульном тестировании. Интеграционные тесты обычно имитируют вызовы сервисов и изменение состояния компонентов пользовательского интерфейса. В примере 6.2 имитируется ответ на попытку входа, и тест проверяет исчезновение с экрана формы входа после успешного входа. Также интеграционные тесты могут помочь проверить компоненты с несколькими дочерними компонентами и интеграцию между ними в разных состояниях.

Пример 6.2. Пример интеграционного теста с использованием Jest

```
test('User is able to login successfully', async () => {

  // имитация ответа на попытку входа
  jest
    .spyOn(window, 'fetch')
    .mockResolvedValue({ json: () => ({ message: 'Success' }) });

  render(<LoginForm />);
```

```
const emailInput = screen.getByLabelText('Email');
const passwordInput = screen.getByLabelText('Password');
const submit = screen.getByRole('button');

// ввод учетных данных и отправка формы
fireEvent.change(emailInput, { target: { value: 'testUser@mail.com' } });
fireEvent.change(passwordInput, { target: { value: 'admin123' } });
fireEvent.click(submit);

// кнопка отправки должна немедленно деактивироваться
expect(submit).toBeDisabled();

// ждать сокрытия элементов формы после успешного входа
await waitFor(() => {
  expect(submit).not.toBeInTheDocument();
  expect(emailInput).not.toBeInTheDocument();
  expect(passwordInput).not.toBeInTheDocument();
});
});
```

Разработчики пишут эти тесты по завершении разработки компонента и хранят их вместе с прикладным кодом. Подобно модульным тестам, они обеспечивают быструю обратную связь на этапе разработки и способствуют визуальному тестированию, как, например, в этом примере, подтверждая исчезновение соответствующих элементов после входа в систему. Для интеграционного тестирования можно использовать те же инструменты, что и для модульного тестирования. Кроме того, рекомендуется добавлять тесты доступности на уровне компонентов.

Скриншот-тесты

Скриншот-тесты предназначены для проверки структурных аспектов отдельных компонентов и их групп, что напрямую способствует визуальному тестированию на микроуровне. Эти тесты визуализируют фактическую структуру DOM компонентов с помощью средств тестирования и сравнивают результаты с ожидаемой структурой, хранящейся в эталонном скриншоте вместе с тестом. Для этой цели, например, можно использовать такие инструменты, как Jest (<https://jestjs.io/docs/snapshot-testing>) и react-test-renderer (<https://reactjs.org/docs/test-renderer.html>).



Скриншот-тесты сравнивают фрагменты HTML-кода. Они отличаются от визуальных тестов, обсуждаемых далее, которые попиксельно сравнивают изображения (скриншоты).

В примере 6.3 показан тест скриншота для проверки структуры DOM компонента `Link` с помощью Jest.

Пример 6.3. Пример скриншот-теста с использованием Jest

```
import React from 'react';
import renderer from 'react-test-renderer';
import Link from '../Link.react';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link page="http://www.example.com">Sample Site</Link>)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

Для каждого коммита кода этот тест создает новый файл со структурой DOM компонента `Link`, как показано в примере 6.4, и сверяет его с предыдущим скриншотом.

Пример 6.4. Файл со скриншотом, созданный Jest

```
exports[`renders correctly`] = `
<a
  className="test"
  href="http://www.example.com"
  onMouseEnter={[Function]}
  onMouseLeave={[Function]}
>
  Sample Site
</a>
`;
```

Эти тесты позволяют быстро получить информацию о структурных аспектах компонентов. (Визуальные тесты, напротив, требуют, чтобы приложение было полностью функциональным.) Данные тесты становятся особенно важными, когда компоненты повторно используются в нескольких приложениях, например в системах проектирования. Подобно модульным и интеграционным тестам, они пишутся в процессе разработки и хранятся вместе с прикладным кодом.

Рекомендуется сужать направленность скриншот-тестов, например тестировать небольшие компоненты, такие как кнопки или заголовки, по одному или более крупные компоненты, которые не предполагается менять часто. Лучше всего писать их после разработки компонентов для регрессионного тестирования. В противном случае могут потребоваться дополнительные усилия для их поддержания даже при незначительных изменениях структуры.

Функциональные сквозные тесты

Как обсуждалось в главе 3, автоматизированные функциональные тесты имитируют действия реального пользователя на веб-сайте в реальном браузере. Их пишут для проверки пользовательских сценариев с учетом интеграции с внешними

и внутренними сервисами. В отличие от тестов, обсуждавшихся ранее, автоматизированные функциональные тесты требуют, чтобы приложение было полностью развернуто и настроено с помощью соответствующих тестовых данных. Хотя эти тесты используют настоящий браузер, они лишь частично способствуют визуальному тестированию, потому что проверяют наличие элемента по его локатору, но не контролируют его внешний вид.

Визуальные тесты

Все обсуждавшиеся ранее типы тестов так или иначе помогают проводить визуальное тестирование, но основную работу выполняют, конечно, визуальные тесты. Как и функциональные тесты, описанные в предыдущем разделе, они открывают приложение в браузере, затем сравнивают скриншот (снимок экрана) каждой страницы с эталонным скриншотом. Визуальные тесты можно хранить отдельно или интегрировать в набор функциональных тестов, чтобы их проще было поддерживать. Для этой цели можно использовать инструменты с открытым исходным кодом, такие как Cypress, Galen, BackstopJS и др., а также коммерческие инструменты, например Applitools Eyes, CrossBrowserTesting и Percy.

ВИЗУАЛЬНОЕ ТЕСТИРОВАНИЕ ПО СРАВНЕНИЮ СО СКРИНШОТАМИ

Визуальные тесты и скриншот-тесты могут показаться похожими, но они работают на разных уровнях. Их можно сравнить с высокоуровневыми функциональными сквозными тестами и низкоуровневыми тестами API. Цикл обратной связи этих двух видов тестов также существенно различается. И как уже говорилось, визуальные тесты проверяют приложение после того, как оно полностью отображается в браузере, тогда как скриншот-тесты дают обратную связь о структуре HTML и, следовательно, удобны для разработчиков и помогают при раннем тестировании.

Скриншот-тесты хорошо подходят для случаев, когда они сосредоточены на отдельных компонентах небольшого размера, тогда как визуальные тесты идеальны для проверки интеграции нескольких компонентов в более широком контексте, таком как веб-страница.

Кросс-браузерное тестирование

Кросс-браузерное тестирование преследует две важные цели: выполнение функциональной проверки и проверки визуального качества в разных браузерах. Функциональный поток приложения обычно почти не меняется в разных браузерах, но были случаи выявления расхождений. Например, в 2020 году Twitter пришлось исправлять критическую проблему безопасности (<https://oreil.ly/hG81i>), из-за которой конфиденциальная информация пользователей сохранялась в кэше браузера

Firefox. Судя по всему, в Chrome такой проблемы не было. Поэтому тестирование функционального потока в разных браузерах должно быть частью стратегии кросс-браузерного тестирования.

Приступая к такому тестированию, прежде всего необходимо определиться со списком браузеров, на которых вы собираетесь сосредоточиться. Как мы видели ранее, наиболее широкое распространение в мире получили браузеры Chrome и Safari, и пользователи могут обращаться к вашему приложению с их помощью, используя различные устройства: настольные ПК, планшеты и смартфоны. Еще один важный фактор, который следует учитывать при тестировании в разных браузерах, — скорость реагирования приложения. Общее практическое правило — сосредоточиться на браузерах и разрешениях, на которые приходится 80 % пользователей. Оставшиеся 20 % можно протестировать в ходе проверки ошибок ближе к выпуску.

Таким образом, хорошей стратегией получения обратной связи о совместимости с разными браузерами, учитывая замечания, сделанные при обсуждении функциональных тестов на основе пользовательского интерфейса (предполагающих более медленную обратную связь и не включающих визуальное тестирование), может стать выбор ограниченного числа наиболее важных функциональных потоков и их опробование в выбранных вами браузерах. А для получения обратной связи о качестве изображения можно повторно применять визуальные тесты. Они способны дать информацию как о совместимости с браузерами, так и об отзывчивости приложения. Выберите браузеры и разрешения экрана, используемые 80 % ваших конечных пользователей, и добавьте визуальные тесты для наиболее важных пользовательских сценариев. В итоге у вас должно быть несколько функциональных и визуальных тестов (их можно объединить с помощью таких инструментов, как Cypress и Applitools Eyes), проверяющих совместимость с браузерами и скорость реагирования приложения.

Если вам кажется, что этих усилий будет недостаточно для кросс-браузерного тестирования всех страниц приложения, то задействуйте инструменты/библиотеки разработки пользовательского интерфейса, такие как React, Vue.js, Bootstrap и Tailwind, имеющие встроенную поддержку проверки совместимости. Они могут помочь обеспечить визуальное качество некритичных пользовательских сценариев в приложении. Но имейте в виду, что эти инструменты и библиотеки поддерживают только наиболее свежие стандартизированные версии браузеров и некоторые их возможности могут не поддерживаться старыми браузерами.

Проверить, поддерживает ли данный браузер определенную функцию среды разработки (и, следовательно, возможность ее использования), можно по таблицам поддержки на сайте *CanIUse* (<https://caniuse.com/ciu/comparison>). Например, если вы хотите применить макет *flexbox* CSS в своем пользовательском интерфейсе, то уточните сначала, поддерживают ли его целевые браузеры. Для автоматической

проверки функций CSS, не поддерживаемых целевыми браузерами, на основе данных CanIUse можно использовать также плагины, такие как `stylelint-no-unsupported-browser-features` (<https://oreil.ly/Zo62P>). Похожий плагин `eslint-plugin-caniuse` (<https://oreil.ly/asdQ1>) поможет выявить функции сценариев, не поддерживаемые целевыми браузерами. Существует еще один способ обеспечить обратную совместимость кода JavaScript — применять транспилеры, такие как Babel. Они преобразуют код, написанный на последней версии JavaScript, в версию, совместимую со старыми браузерами. Используя все перечисленные инструменты, вы сможете гарантировать соответствие всех страниц вашего приложения требованиям совместимости с разными браузерами, особенно с точки зрения их визуального качества.

РАННЕЕ КРОСС-БРАУЗЕРНОЕ ТЕСТИРОВАНИЕ

Начиная слева:

- применяйте библиотеки разработки, такие как React, Vue.js и др., которые поддерживают стандартизированные браузеры;
- задействуйте плагины и инструменты, такие как `stylelint-no-unsupported-browser-features` и `CanIUse`, чтобы обеспечить совместимость функций пользовательского интерфейса с целевыми браузерами во время разработки;
- создайте несколько функциональных тестов на основе пользовательского интерфейса и визуальных тестов для проверки совместимости с выбранным набором браузеров и устройств, охватывающих 80 % потенциальных пользователей вашего приложения;
- регулярно проверяйте наличие ошибок, чтобы охватить как можно большую часть из оставшихся 20 %.

Тестирование производительности фронтенда

Тестирование производительности фронтенда включает проверку задержек при отрисовке визуальных компонентов браузером. Вы можете повысить привлекательность и визуальное качество приложения, добавив приятные глазу изображения и необычные жесты, но если это приведет к ухудшению производительности, то пользователи, скорее всего, уйдут от вас. Общеизвестно, что загрузка компонентов внешнего интерфейса занимает около 80 % всего времени загрузки страницы. Поэтому очень важно найти правильный баланс между производительностью интерфейса и качеством изображения. Инструменты и передовые приемы тестирования производительности фронтенда подробно обсуждаются в главе 8, тем не менее, учитывая их относительную важность, они заслужили упоминания и здесь.

Тестирование доступности

Законодательство многих стран предусматривает нормы доступности веб-сайтов, поэтому код пользовательского интерфейса должен разрабатываться в соответствии с требованиями WCAG 2.0 (<https://oreil.ly/TRxmX>). Функции доступности значительно влияют на визуальное качество веб-сайта и в целом улучшают его, поскольку поощряют единообразие макета организации всего сайта, наличие понятного текста, достаточного пространства для щелчков кнопкой мыши и т. д. Подробнее об инструментах тестирования доступности и передовых практиках рассказывается в главе 9.

На этом мы завершаем обзор различных видов тестирования фронтенда. Главный вывод, который можно сделать: команда должна адаптировать свою стратегию тестирования фронтенда к целям различных видов тестов и потребностям приложения. В целом рекомендуется иметь больше тестов на микроуровне (например, модульных) и меньше — на макроуровне (например, визуальных и сквозных функциональных).

Упражнения

Теперь вы готовы приступить к изучению инструментов автоматизированного визуального тестирования. Для этого можно применять инструменты, поддерживающие управление из командной строки или из программного кода либо делегировать эту задачу поставщикам программного обеспечения как услуги (Software-as-a-Service, SaaS). Здесь мы рассмотрим два упражнения с использованием BackstopJS и Cypress. Можете добавить эти визуальные тесты в свои конвейеры CI и запускать их после развертывания каждого коммита вместе с функциональными тестами.

BackstopJS

BackstopJS (<https://github.com/garris/BackstopJS>) — популярный инструмент визуального тестирования, который поддерживается активным сообществом разработчиков ПО с открытым исходным кодом. По сути это библиотека Node, которую легко интегрировать с CI, использующая стиль тестирования на основе конфигурации, что означает отсутствие необходимости писать программный код. Она применяет *Puppeteer*, инструмент автоматизации пользовательского интерфейса, для отображения приложения в Chrome и навигации по нему, а также *Resemble.js* для сравнения скриншотов веб-страниц. После сравнения изображений BackstopJS генерирует результаты в форме HTML-отчетов. Она позволяет настраивать чувствительность сравнения изображений и автоматического исправления после неудачных тестов, о чем мы сейчас и поговорим.

Это упражнение поможет вам создать визуальный тест с помощью BackstopJS для проверки веб-приложения в трех разрешениях, типичных для планшетов, мобильных телефонов и обычных браузеров.

Установка и настройка

Предварительно вам может потребоваться установить и настроить Node.js и Visual Studio Code (как в упражнении Cypress в главе 3). После этого выполните следующие действия, чтобы установить инструмент и получить базовую структуру проекта.

1. Создайте новую папку проекта и запустите следующую команду в терминале, чтобы установить BackstopJS:

```
$ npm install -g backstopjs
```

Библиотека BackstopJS будет установлена глобально на вашем локальном компьютере, чтобы ее можно было повторно использовать в разных проектах. Эта команда попутно установит движки `chromium` (для поддержки Chrome) и `puppeteer`.

2. Настройте конфигурации по умолчанию и структуру проекта с помощью следующей команды:

```
$ backstop init
```

Теперь вы сможете просмотреть файл конфигурации по умолчанию `backstop.json` в папке проекта. В него будете добавлять свои визуальные тесты в форме конфигураций.

Рабочий процесс

Теперь возьмите любой образец общедоступного веб-сайта для визуального тестирования и выполните следующие действия, чтобы создать тест.

1. Чтобы проверить веб-страницу в трех разных разрешениях экрана, как того требует наш тестовый сценарий, добавьте конфигурацию `backstop.json`, показанную в примере 6.5.

Пример 6.5. Пример теста в конфигурационном файле `backstop.json`

```
{
  "id": "backstop_demo",
  "viewports": [
    {
      "label": "browser",
      "width": 1366,
      "height": 784
    },
  ],
}
```

```

    {
      "label": "tablet",
      "width": 1024,
      "height": 768
    },
    {
      "name": "phone",
      "width": 320,
      "height": 480
    }
  ],
  "onBeforeScript": "puppet/onBefore.js",
  "onReadyScript": "puppet/onReady.js",
  "scenarios": [
    {
      "label": "Application Home page",
      "cookiePath": "backstop_data/engine_scripts/cookies.json",
      "url": "<укажите здесь URL приложения>",
      "referenceUrl": "<укажите здесь тот же самый URL>",
      "readyEvent": "",
      "delay": 5000,
      "hideSelectors": [],
      "removeSelectors": [],
      "hoverSelector": "",
      "clickSelector": "",
      "readySelector": "",
      "postInteractionWait": 0,
      "selectors": [],
      "selectorExpansion": true,
      "expect": 0,
      "misMatchThreshold": 0.1,
      "requireSameDimensions": true
    }
  ],
  "paths": {
    "bitmaps_reference": "backstop_data/bitmaps_reference",
    "bitmaps_test": "backstop_data/bitmaps_test",
    "engine_scripts": "backstop_data/engine_scripts",
    "html_report": "backstop_data/html_report",
    "ci_report": "backstop_data/ci_report"
  },
  "report": ["browser"],
  "engine": "puppeteer",
  "engineOptions": {
    "args": ["--no-sandbox"]
  },
  "asyncCaptureLimit": 5,
  "asyncCompareLimit": 50,
  "debug": false,
  "debugWindow": false
}

```

Вот несколько важных замечаний, касающихся этого файла.

- Массив `viewports` определяет три разрешения экрана, типичные для настольных компьютеров, планшетов и мобильных устройств.
 - Сценарии Puppeteer для взаимодействия с элементами пользовательского интерфейса в Chrome настраиваются с помощью параметров `onBeforeScript` и `onReadyScript`. Также можно добавить свои сценарии для определения новых действий.
 - Тестовый сценарий определяется в массиве `scenarios` с такими параметрами, как `url`, `referenceURL`, `clickSelector`, `hideSelectors` и т. д. Позже вы узнаете назначение каждого из них.
 - Места хранения эталонных и тестовых скриншотов определяются параметрами `bitmaps_reference` и `bitmaps_test`. Место для сохранения отчетов задается параметром `html_report`.
 - Параметру `report` присвоено значение `"browser"`, чтобы разрешить просмотр результатов в браузере. Если ему присвоить значение `"CI"`, то будет сгенерирован отчет в формате JUnit.
 - Параметр `engine` применяется для настройки соответствующего браузера. По умолчанию ему присваивается значение `"puppeteer"`, соответствующее механизму, который использует браузер Chrome. Вы можете изменить это значение на `"phantomjs"`, чтобы запускать тесты в Firefox с помощью более старых версий BackstopJS.
 - Параметр `asyncCaptureLimit` со значением 5 будет запускать тесты параллельно в пяти потоках.
2. Следующий шаг — создание эталонных скриншотов веб-страницы на экранах разных размеров для сравнения. Библиотека BackstopJS поможет вам в этом и выполнит все автоматически. Она откроет URL, указанный в параметре `referenceURL`, сделает эталонные снимки с разными размерами экрана, которые перечислены в массиве `viewports`, и сохранит их в папке, указанной в параметре `bitmaps_reference`. Вот команда, которая делает это:

```
$ backstop reference
```

3. Следующий шаг — запуск тестов с помощью команды:

```
$ backstop test
```

После ее запуска BackstopJS проверит веб-сайт, заданный параметром `url`, и сравнит его внешний вид с эталонными скриншотами для всех разрешений.

По завершении теста вы сможете просмотреть результаты в браузере (рис. 6.4). В качестве тестового сайта я выбрала домашнюю страницу Amazon, и, как показывают результаты, один тест успешно пройден, а два потерпели неудачу. В частности, тест в браузере с большим разрешением экрана пройден, а два других — нет.

Открыв отчет о неудачных тестах, я увидела эталонные и реальные скриншоты, кроме того, там есть и третье изображение, подчеркивающее различия с первыми двумя. Все три изображения приведены на рисунке. Различия выделены в нижней половине третьего изображения.

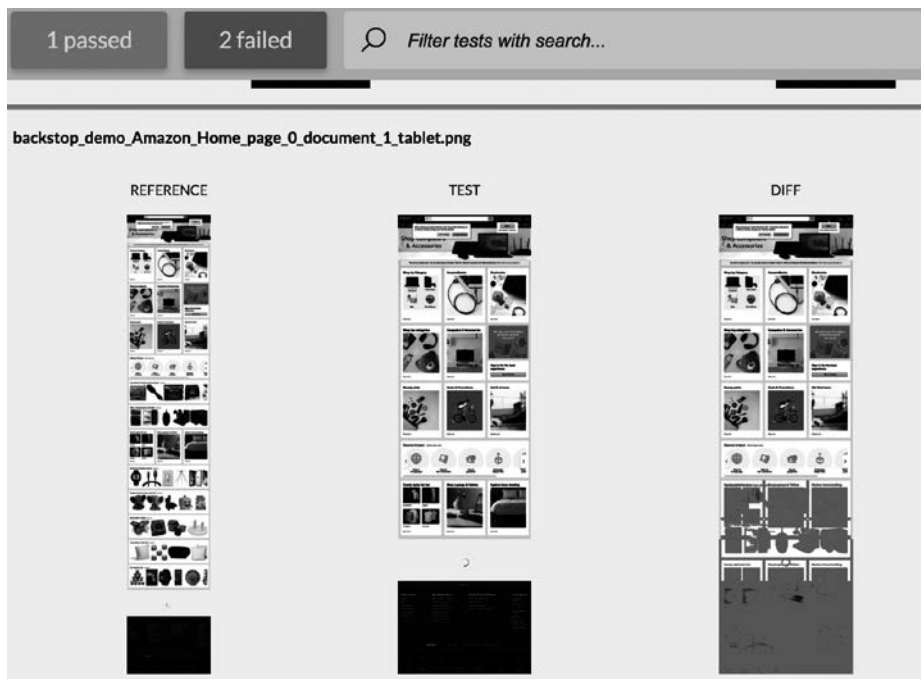


Рис. 6.4. Отчет BackstopJS по результатам тестирования домашней страницы Amazon

Неудачи обусловлены динамическим контентом на домашней странице Amazon. В тестовой среде есть возможность контролировать тестовые данные, но если вы столкнетесь с динамическим контентом в своем приложении, то BackstopJS позволит скрыть его во время выполнения теста. Скрыть или полностью удалить динамическое содержимое можно с помощью параметров `hideSelectors` или `removeSelectors` в файле `backstop.json`. В роли селекторов могут выступать имена классов или идентификаторов, как показано здесь:

```
"hideSelectors": [".feed-carousel-viewport"]
```

Также можно выбрать определенные компоненты на экране для визуального тестирования и исключить динамический контент, используя параметр `selectors`.

Иногда даже после удаления динамического контента тесты могут терпеть неудачу из-за небольших изменений на уровне пикселей, которые на самом деле не ухуд-

шают визуальное качество. В таких случаях можно настроить чувствительность тестов в параметре `misMatchThreshold`, задав значение от 0,00 до 100,00 %. Это может избавить вас от хаоса при тестировании и обслуживании.

BackstopJS помогает и при сопровождении тестирования. Представьте, что ваше приложение изменилось и нужно обновить эталонные скриншоты. Вы можете просто одобрить новые скриншоты, сделанные во время последнего тестирования (конечно же, предварительно просмотрев их глазами), с помощью следующей команды:

```
$ backstop approve
```

Также можно расширить тест и реализовать в нем поиск товара и проверку страницы с информацией о нем с помощью `keyPressSelectors`. В примере 6.6 показана конфигурация, настраивающая ввод текста в поле поиска на сайте Amazon и нажатие кнопки поиска.

Во многих проектах часто сравниваются страницы в разных средах, например на локальном компьютере и в среде тестирования. Это можно сделать, указав локальный URL в параметре `url` и URL тестовой среды — в параметре `referenceURL`.

Пример 6.6. Ввод текста для поиска с помощью `keyPressSelectors` в конфигурационном файле `backstop.json`

```
"keyPressSelectors": [
  {
    "selector": "#twotabsearchtextbox",
    "keyPress": "Women's Tshirt"
  }
],
"clickSelectors": ["#nav-search-submit-button"],
```

При интеграции с CI нужно изменить значение параметра `report` на "CI" и сохранить все результаты вывода, включая скриншоты. Также рекомендуется заархивировать старые скриншоты, чтобы при необходимости можно было изучить историю.

Cypress

Предварительные условия и настройку среды автоматизации функционального тестирования с использованием Cypress мы обсудили в главе 3. Вы можете организовать визуальное тестирование как часть той же среды с помощью плагина `cypress-plugin-snapshots` (<https://oreil.ly/76YwA>). Как и BackstopJS, он сравнивает скриншоты и выделяет различия между ними. А также предоставляет похожие возможности, например позволяет настраивать чувствительность теста, выбирать элементы для сравнения и делать многое другое.

Установка и настройка

Чтобы начать работу с плагином Cypress, выполните следующие действия.

1. Для установки плагина запустите команду:

```
$ npm i cypress-plugin-snapshots -S
```

2. В файлы `cypress/plugins/index.js` и `cypress/support/index.js` добавьте код, импортирующий команды плагина, как показано в примере 6.7.

Пример 6.7. Конфигурация плагина Cypress

```
// cypress/plugins/index.js

const { initPlugin } = require('cypress-plugin-snapshots/plugin');

module.exports = (on, config) => {
  initPlugin(on, config);
  return config;
};

// cypress/support/index.js

import 'cypress-plugin-snapshots/commands';
```

3. Конфигурация Cypress хранится в файле `cypress.json`. Добавьте в него тестовые конфигурации, как показано в примере 6.8. Отмечу несколько важных параметров: `threshold` определяет чувствительность теста, `auto clean-up` автоматически удаляет неиспользуемые скриншоты и `ignoreFields` — массив компонентов, исключаемых из сравнения скриншотов.

Пример 6.8. Пример теста в конфигурационном файле `cypress.json`

```
{
  "env": {
    "cypress-plugin-snapshots": {
      "autoCleanUp": false,
      "autopassNewSnapshots": true,
      "diffLines": 3,
      "excludeFields": [],
      "ignoreExtraArrayItems": false,
      "ignoreExtraFields": false,
      "normalizeJson": true,
      "prettier": true,
      "imageConfig": {
        "createDiffImage": true,
        "resizeDevicePixelRatio": true,
        "threshold": 0.01,
        "thresholdType": "percent"
      }
    }
  },
}
```

```

"screenshotConfig": {
  "blackout": [],
  "capture": "fullPage",
  "clip": null,
  "disableTimersAndAnimations": true,
  "log": false,
  "scale": false,
  "timeout": 30000
},
"serverEnabled": true
"serverHost": "localhost",
"serverPort": 2121,
"updateSnapshots": false,
"backgroundBlend": "difference"
}
}}

```

Рабочий процесс

Для добавления визуальных тестов плагин Cypress предоставляет метод `toMatchImageSnapshot()`, который делает скриншот указанного компонента или текущей страницы и сравнивает его с базовым скриншотом. В качестве базовых он использует скриншоты, сделанные во время первого запуска теста. В примере 6.9 показан тест, который открывает URL приложения, ждет, пока страница станет видимой, а затем делает скриншот всего ее содержимого для сравнения изображений.

Пример 6.9. Визуальный тест с использованием Cypress для проверки домашней страницы приложения

```

describe('Application Home page', () => {
  it('Visits the Application home page', () => {
    cy.visit('<укажите здесь URL приложения>')
    cy.get('#twotabsearchtextbox')
      .should('be.visible')
    cy.get('#pageContent').toMatchImageSnapshot()
  })
})

```

Если запустить тест на домашней странице Amazon, он завершится неудачей из-за динамического контента. Результаты тестирования в Cypress с выделенными различиями в изображениях показаны на рис. 6.5.

Базовый, эталонный и сравниваемый скриншоты с выделенными различиями находятся в отдельной папке и могут сохраняться в CI для отладки. Преимущества объединения визуальных и функциональных тестов заключаются в более простом обслуживании и возможности повторного использования сценариев создания тестовых данных.

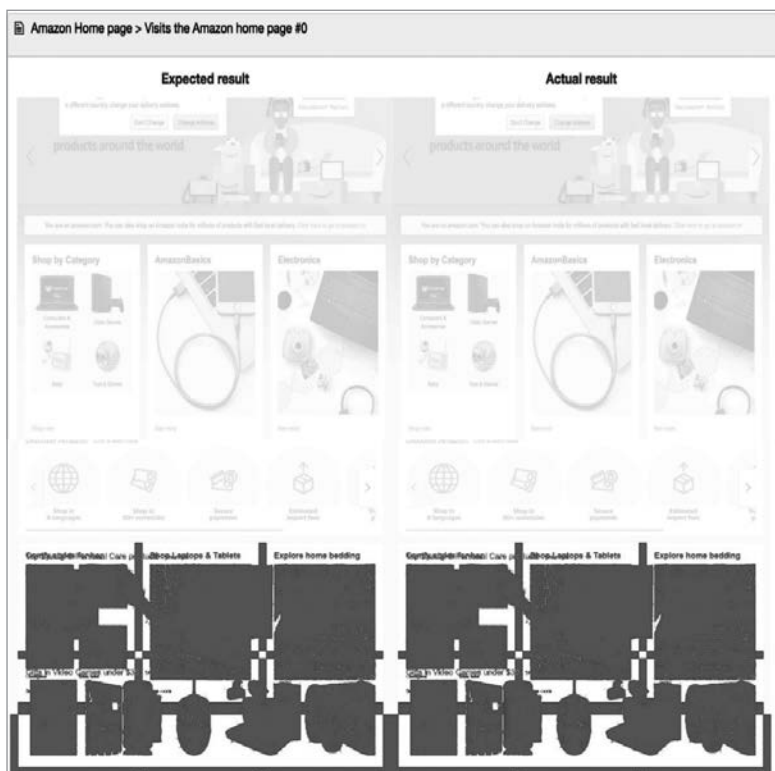


Рис. 6.5. Результаты сравнения скриншотов домашней страницы Amazon в Cypress

Дополнительные инструменты тестирования

Как упоминалось ранее, существует множество методов визуального тестирования. Я познакомлю вас еще с парой инструментов, которые можно использовать для добавления автоматизированных визуальных тестов в свое приложение, чтобы вы могли получить более широкое представление о сфере визуального тестирования.

Applitools Eyes, инструмент на базе искусственного интеллекта

ИИ, проникший в сферу визуального тестирования, использует компьютерное зрение и технологии глубокого обучения. Компьютерное зрение — это область искусственного интеллекта, которая позволяет компьютерам просматривать цифровой медиаконтент, например изображения и видео, и исследовать его. С инженерной точки зрения цель состоит в том, чтобы заставить компьютеры изучать и решать задачи, которые может выполнять зрительная система человека, например просматривать веб-страницы и оценивать изменения.

Applitools Eyes (<https://oreil.ly/n1hc8>) — один из таких инструментов визуального тестирования на базе искусственного интеллекта. Технология компьютерного зрения, на которой он основан, называется Visual AI и обладает когнитивными способностями, позволяющими анализировать структуру и макет страницы, включая цвета и формы элементов, и выявлять различия так же, как это сделал бы человеческий глаз. Предлагается как коммерческое SaaS-решение.

Модель Visual AI обучена преодолевать некоторые распространенные препятствия, возникающие при визуальном тестировании, например такие.

Обслуживание

Когда тесты терпят неудачу из-за типичных визуальных изменений, она распознает их и при одобрении автоматически исправляет базовые скриншоты, если необходимо.

Динамическая обработка данных

В отличие от точного попиксельного сравнения, она может опускать динамические данные.

Контроль чувствительности

Чувствительность можно настроить так, чтобы игнорировать незначительные изменения пользовательского интерфейса, которые не имеют значения.

Чтобы настроить Eyes, необходимо зарегистрироваться в Applitools и получить закрытый ключ API для доступа к серверу Eyes, размещенному в облаке. Реальное сравнение выполняет сервер Eyes. Вам потребуется загрузить комплект инструментов разработки (Software Development Kit, SDK), чтобы установить соединение с сервером и настроить его с помощью ключа API. Eyes SDK предоставляет соответствующие API для захвата и отправки скриншотов на сервер Eyes. Эти API можно использовать в существующих тестах Selenium WebDriver для визуального тестирования. (SDK также интегрируется со многими другими инструментами разработки и тестирования пользовательского интерфейса, такими как Cypress, React Storybook и даже Arrium — инструментом автоматизации тестирования приложений на мобильных устройствах.)

Вот основные API из Eyes SDK, которые вам пригодятся в тестах:

- `eyes.open(driver)` — создает экземпляр соединения между экземпляром WebDriver и сервером Eyes;
- `eyes.checkWindow()` — проверяет визуальное качество страницы на всех указанных устройствах и браузерах;
- `eyes.closeAsync()` — сообщает серверу Eyes, что тест завершен, и тот должен сгенерировать результаты.

В примере 6.10 показан фрагмента теста Selenium WebDriver, использующий Eyes API.

Пример 6.10. Интеграция Applitoools Eyes с тестами WebDriver

```
// Визуальная контрольная точка 1 после перехода
// на домашнюю страницу приложения

driver.get("<укажите здесь URL приложения>");
eyes.checkWindow("Application Homepage");

// Визуальная контрольная точка 2 после нажатия кнопки

driver.findElement(By.className("searchbutton")).click();
eyes.checkWindow("After clicking search button on home page");
```

Applitoools Eyes обеспечивает увеличение производительности за счет создания моментального снимка DOM веб-страницы (вместо скриншота) при запуске тестов Selenium WebDriver и его использования для параллельного сравнения веб-страницы в нескольких браузерах, устройствах и с разными разрешениями экрана, поддерживаемыми облачной инфраструктурой Applitoools. Это не только обеспечивает сверхвысокую производительность, но и экономит затраты на инфраструктуру тестирования вашего проекта, поскольку все устройства размещаются в облаке. Продукт также предоставляет панель мониторинга для управления общим рабочим процессом.

Storybook

Storybook (<https://storybook.js.org>) — это инструмент с открытым исходным кодом, помогающий при разработке фронтенда и довольно популярный (около 70 тыс. звезд на GitHub) в мире разработки фронтенда. Он хорошо интегрируется с различными фреймворками и библиотеками, такими как React, Vue.js и Angular. Это помогает создавать компоненты пользовательского интерфейса изолированно и разрабатывать весь пользовательский интерфейс без настройки сложного стека приложений, создания тестовых данных или навигации по приложению.

Storybook дает разработчикам возможность создавать новые компоненты и вручную проверять их поведение и внешний вид, визуализируя внутри самого инструмента. Точно так же с его помощью можно проверить различные состояния компонента. Storybook сохраняет историю визуализации компонентов в разных состояниях. Например, компонент кнопки может отображаться в разных состояниях, таких как «большая», «маленькая» и т. д., и Storybook будет сохранять каждое из них в виде отдельной истории. Такие истории служат отличной основой для визуального тестирования.

Фактически инструмент предлагает визуальное тестирование «из коробки» через Chromatic (<https://www.chromatic.com>) — интернет-сервис, дополненный разработчиками Storybook поддержкой автоматизированного визуального тестирования в нескольких браузерах (имеется бесплатный тариф с некоторыми ограничениями). Вы можете использовать Chromatic для автоматического сравнения

каждой новой истории с предыдущей — это действительно раннее визуальное тестирование.

В организациях, где централизованная группа разработчиков пользовательского интерфейса создает общие компоненты, не требующие взаимодействия с сервером, этот инструмент очень пригодится.

Как видите, существует множество способов визуального тестирования, которые можно интегрировать в процесс разработки. Визуальное тестирование можно интегрировать в среду разработки с помощью Storybook, в процесс разработки с помощью BackstopJS или в функциональные тесты с помощью Cypress и Applitools Eyes. Все они открывают новые возможности для быстрого получения обратной связи в соответствии с потребностями вашего проекта.

Перспективы: проблемы визуального тестирования

Одна из проблем визуального тестирования — выбор инструментов. В этой главе была перечислена лишь малая их часть. Учитывая наличие поставщиков инструментов на основе искусственного интеллекта и SaaS, выбор оказывается очень широким. Вот некоторые особенности, на которые следует обращать внимание при выборе инструмента автоматизации визуального тестирования.

- Простота всего рабочего процесса от создания тестов до обслуживания и интеграции с CI.
- Надежные методы управления скриншотами. Если для каждого небольшого изменения вам потребуется заменять сотни базовых изображений, то затраты на визуальное тестирование окажутся поистине огромными. Инструменты, предлагающие автоматическое обновление скриншотов, — большое преимущество.
- Контроль чувствительности. Инструмент должен давать возможность игнорировать незначительные изменения в пользовательском интерфейсе.
- Умение обрабатывать динамические данные.
- Возможность запуска в разных браузерах и имитации различных устройств.
- Высокая производительность при выполнении визуальных тестов в различных сочетаниях браузеров и устройств.

В общем, вам потребуется не только выбрать инструменты, но и заставить товарищей по команде поверить в идею автоматизированного визуального тестирования, потому что для создания и поддержки этих тестов нужно приложить дополнительные усилия. Если сделать это вовремя и выбрать инструменты, упрощающие тестирование, то ваша команда вскоре оценит эту возможность. Тем не менее помните, что не все приложения нуждаются в автоматизированном визуальном тестировании. Прежде чем встать на этот путь, исследуйте варианты использования вашего проекта и проанализируйте затраты и выгоды.

Ключевые выводы

- Визуальное тестирование дает возможность убедиться в том, что приложение выглядит в полном соответствии с замыслом. Визуальное качество приближает бизнес к завоеванию доверия клиентов, что, в свою очередь, повышает ценность бренда компании.
- Ручное визуальное тестирование и функциональные тесты на основе пользовательского интерфейса могут частично способствовать визуальному тестированию, но их одних недостаточно, потому что ручное тестирование чревато ошибками, а функциональные тесты не проверяют визуальные аспекты приложения. Следовательно, вам могут потребоваться отдельные автоматизированные визуальные тесты.
- Автоматизированные визуальные тесты могут принести большую пользу в зависимости от характера приложения и объема работы. Подумайте о таких факторах, как влияние на опыт клиентов, уверенность команды, объемы и виды ручных работ, и определите, нужны ли вашему приложению автоматизированные визуальные тесты.
- Инструменты с открытым исходным кодом, такие как BackstopJS, Storybook и Cypress, обеспечивают разнообразные возможности для автоматизации визуального тестирования. Решения SaaS, такие как Applitoools Eyes и Chromatic, предлагают дополнительные функции управления инфраструктурой и рабочими процессами за определенную плату.
- Применяйте визуальные тесты на правильных этапах разработки приложения, чтобы избежать нестабильности, и выбирайте инструменты, которые смогут дать быструю и стабильную обратную связь на ранних этапах цикла поставки.
- Визуальное тестирование — это лишь часть экосистемы тестирования пользовательского интерфейса. Применение других видов тестирования пользовательского интерфейса на микро- и макроуровне и разработка хорошей стратегии тестирования может помочь быстрее получить обратную связь по визуальному качеству.

Тестирование безопасности

Прочность цепи определяется прочностью самого слабого ее звена.

Томас Рид. Опыты об интеллектуальных способностях человека

В настоящее время нам, особенно имеющим учетные записи в социальных сетях, как никогда прежде, грозит то, что против нас будут совершены киберпреступления! *Киберпреступность* — это общий термин, обозначающий любые незаконные действия, осуществляемые с помощью компьютера и сетей, включая кражу финансов, частных активов, таких как торговые документы и отчеты об исследованиях, использование конфиденциальной информации, такой как биологические данные человека, и т. д. По оценкам экспертов по кибербезопасности, к 2025 году ежегодные глобальные потери из-за киберпреступлений (прямые и косвенные) вырастут до 10,5 трлн долларов США (<https://oreil.ly/OwtEm>) с примерно 6 трлн в 2021 году. Львиная доля этой суммы приходится на киберпреступления, совершаемые с помощью социальных сетей: согласно исследованию 2019 года, глобальный годовой доход преступников оценивается в 3,25 млрд долларов (<https://oreil.ly/G4zyD>). Это, несомненно, огромные суммы, и, к сожалению, их часть могут составлять деньги, честно заработанные нами и нашими друзьями!

Цифры показывают, что киберпреступления распространены гораздо шире, чем можно себе представить. Заголовки ежедневных новостей (рис. 7.1) доказывают, что киберпреступления не ограничиваются только банковскими сайтами или сайтами социальных сетей, но распространяются на все виды веб-сайтов: атакам подвергаются и сайты бронирования авиабилетов, и сайты знакомств для взрослых. Позже в этой главе мы обсудим примеры таких атак, чтобы получить более полное представление о них. Все это ставит перед разработчиками ПО важный вопрос: какие меры можно предпринять, чтобы защитить приложение от таких атак?

Чтобы получить надежно защищенную систему, рекомендуется строить глубоко эшелонированную защиту, то есть встраивать меры безопасности в несколько

уровней приложения, а не сосредотачиваться на одном внешнем уровне. Это подобно тому, как в давние времена защищали замки: рвом, затем крепкими железными воротами, вооруженной охраной в крепости и т. д. Защита каждого уровня должна быть достаточной для того, чтобы оказывать сопротивление, поскольку каждый уровень, через который злоумышленники прорываются, дает им доступ ко все большему и большему количеству ресурсов.

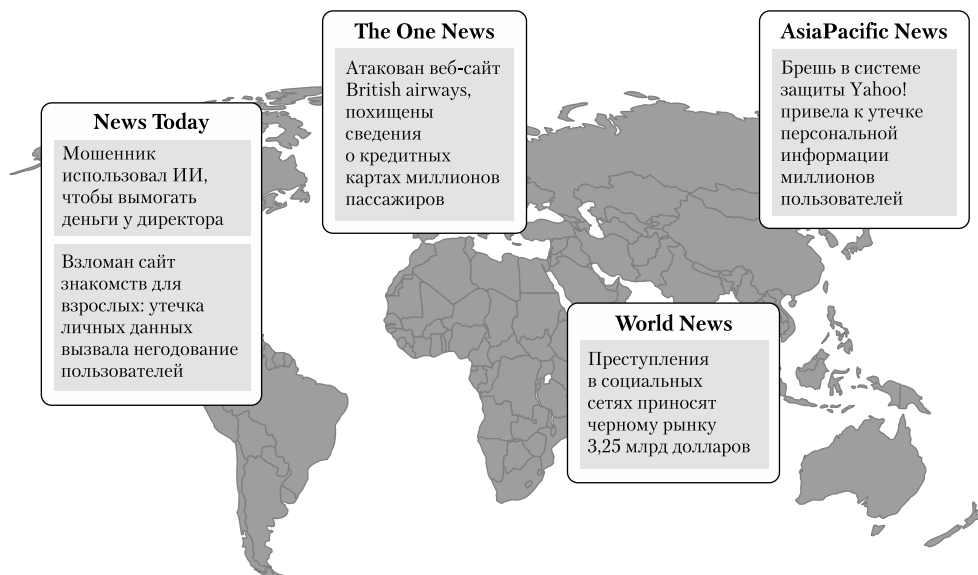


Рис. 7.1. Примеры новостных заголовков, демонстрирующих важность защиты данных

Кроме того, помните, что защита вашей системы настолько надежна, насколько надежна защита самого слабого звена. Поэтому поиск слабых звеньев — это решающий компонент усилий по созданию защищенных, непроницаемых систем. А тестирование безопасности — это основной вид деятельности, который выявляет такие слабые звенья!

Тестирование безопасности требует умения мыслить как хакер и отыскивать потенциальные уязвимости, угрозы и риски в системе, которые могут открыть ворота для киберпреступлений. Профессиональные тестировщики безопасности развивают эти умения в течение многих лет и научились создавать сценарии различных атак на приложение, чтобы выявить уязвимости. Однако вам не обязательно быть таким тестировщиком, чтобы использовать некоторые хорошие инструменты и методы автоматизированного тестирования безопасности в повседневной деятельности для предотвращения серьезных проблем.

Исправление ошибок безопасности, как и любых функциональных ошибок, обходится невероятно дорого, если они обнаруживаются на поздних этапах цикла разработки. Следовательно, чтобы снизить вероятность появления брешей в системах защиты приложения, следует практиковать тестирование безопасности на ранних этапах разработки ПО, а не ждать конца цикла поставки, чтобы привлечь опытных тестировщиков. А это значит, что вы должны начинать тестирование безопасности как можно раньше.

Начинать думать об аспектах безопасности рекомендуется уже на этапе сбора требований. Например, в банковском приложении требование скрывать все транзакции по счету от кого-либо, кроме владельца счета и администратора банка, — очевидная и необходимая функция защиты. Аналогично требование наличия двухфакторной аутентификации обеспечивает дополнительный уровень безопасности, подобно вооруженной охране за каменной стеной, защищающей замок. Использование передовых практик обеспечения безопасности на этапах анализа, разработки и тестирования поможет вам создавать надежно защищенные системы.

В этой главе мы рассмотрим основы, которые необходимо знать, чтобы повысить квалификацию в тестировании безопасности и сдвинуть этот процесс на ранние этапы в цикле поставки. Вы познакомитесь с реальными примерами атак, уязвимостей приложений и с моделью угроз STRIDE. Глава также включает упражнения по моделированию угроз, стратегию тестирования безопасности на ранних этапах разработки ПО, а также пошаговые упражнения, представляющие инструменты тестирования безопасности и рассказывающие, как интегрировать их в конвейеры CI, чтобы получать непрерывную обратную связь о безопасности вашего приложения.



Эта глава не ставит цель подготовить из вас профессиональных тестировщиков безопасности. Как уже говорилось, для этого нужны годы практики. Но это не снимает с разработчиков ПО ответственности за создание защищенных приложений. Поэтому основное внимание здесь уделяется рекомендуемым практикам, инструментам тестирования и, что особенно важно, формированию хакерского мышления.

Введение

Чтобы начать думать как хакер, сначала нужно познакомиться с различными видами кибератак и понять, какие уязвимости способствуют их реализации. Это поможет вам задуматься о потенциальных угрозах для вашего приложения и предотвратить их. Итак, начнем с обсуждения распространенных типов кибератак.

ОСНОВНЫЕ ТЕРМИНЫ, СВЯЗАННЫЕ С БЕЗОПАСНОСТЬЮ

Далее перечислены некоторые термины, которые встретятся вам в этой главе и в других источниках, посвященных вопросам кибербезопасности.

- *Активы (ресурсы)* — критически важные объекты приложения, которые необходимо защищать путем создания соответствующих механизмов.
- *Компрометация* системы защиты происходит, когда защитные механизмы не могут защитить активы.
- *Уязвимости* — потенциальные бреши в системе защиты, которые можно использовать для ее компрометации.
- *Угрозы (факторы риска)* — потенциальные негативные действия или события, которые могут использовать основные уязвимости для компрометации системы защиты.
- *Атака* — несанкционированное вредоносное действие, совершаемое в системе с целью скомпрометировать ее защиту.
- *Шифрование* — метод преобразования информации, после применения которого ее сможет понять только законный получатель, обладающий ключом для расшифровки.
- *Хеширование* — метод преобразования данных любого объема в результат фиксированного размера с помощью алгоритмов (наборов правил для выполнения вычислений или других операций). Полученный результат, или *хеш*, можно использовать для проверки подлинности данных, поскольку даже малейшее их изменение приведет к получению другого хеша. Хеши неизменяемы — хеширование одного и того же уникального набора данных всегда дает один и тот же уникальный результат. Имея хеш, нельзя получить исходное содержимое. Поэтому хеширование считается односторонним методом.

Распространенные кибератаки

В этом разделе описаны наиболее распространенные виды кибератак с примерами из реальной жизни.

Веб-скрапинг

Самый простой способ злоупотребить системой — использовать общедоступные и личные данные пользователей, имеющиеся на веб-сайте. Атака методом веб-скрапинга (web scraping) основана на применении программного обеспечения или сценария, автоматически сканирующего веб-сайт и собирающего информацию, которую можно использовать со злыми намерениями. Основной целью таких атак являются приложения социальных сетей, потому что из них можно выудить много личных данных, таких как местоположение пользователей, номера телефонов и т. д. Для злоумышленника это как если бы вы написали свои данные на плакате и выставили его на улице! По оценкам исследования, упомянутого в на-

чале главы, персональные данные, собранные с сайтов социальных сетей, приносят доход 630 млн долларов в год.

Один из ярких примеров веб-скрапинга был выявлен в 2019 году, когда 419 млн записей пользователей Facebook (<https://oreil.ly/uydzf>), включая номера телефонов, были обнаружены на незащищенном сервере базы данных, доступном из Интернета. Впоследствии Facebook отключил функцию отображения номеров телефонов в профилях пользователей, но злоумышленники уже успели собрать эти сведения. Еще один пример непреднамеренного раскрытия данных был зарегистрирован в 2018-м, когда в Twitter обнаружили ошибку во внутреннем инструменте (<https://oreil.ly/u51SN>), который записывал пароли пользователей в простом текстовом виде без хеширования (вот вам наглядный пример слабого звена!). К счастью, не было найдено никаких признаков компрометации, но в качестве меры предосторожности компания была вынуждена попросить 330 млн своих пользователей сменить пароли.

Открытые данные, размещенные на сайте или где-то еще, всегда дают потенциальную возможность их злонамеренного применения. Если вы думаете как хакер, то ищите такие открытые данные во всем своем приложении.

Метод перебора

Если бы вам понадобилось угадать пароль вашего друга, как бы вы это сделали? Вы могли бы попробовать дату его рождения, любимый цвет, имя супруга или комбинацию из этих сведений, верно? Когда вы расширяете такой метод проб и ошибок, включая в него организованный список всех возможных комбинаций клавиш, это называется атакой методом перебора (brute-force attack).

В 2016 году в результате атаки методом перебора на базы данных FriendFinder Networks было украдено 412 млн записей пользователей (<https://oreil.ly/rwYsc>) с паролями и другой конфиденциальной информацией, такой как сексуальные предпочтения. Сообщается, что имена пользователей и пароли были хешированы с помощью криптографического алгоритма SHA-1, но современные методы перебора и вычислительная мощность смогли преодолеть эту довольно слабую защиту.

Социальная инженерия

Социальная инженерия — это психологическое манипулирование людьми с целью выуживания их конфиденциальной информации. Возможно, вам приходилось сталкиваться с телефонными звонками, когда человек на том конце представлялся работником уважаемой компании и, проявляя неимоверное дружелюбие, пытался выманить у вас сведения о вашей кредитной карте в обмен на какую-то услугу. Если вы поддались на обман, то знайте, что вы не одиноки. В 2019 году на такую удочку попался генеральный директор британской энергетической компании (<https://oreil.ly/6n3Qy>). Ему позвонил телефонный мошенник, голос которого был очень

похож на голос его босса (позже выяснилось, что это была обученная программа искусственного интеллекта), и обманным путем вынудил перевести 243 000 долларов на счет хакера!

Фишинг

Фишинг — это тип атак социальной инженерии, при которой злоумышленник отправляет жертве мошенническое сообщение (обычно электронное письмо) с намерением выкрасть ее личные данные. Целью может быть вынудить загрузить вложение с вредоносным ПО или щелкнуть на ссылке, ведущей на поддельный веб-сайт, очень похожий на настоящий, где жертву попросят ввести учетные данные для входа или данные кредитной карты. Было бы удивительно, если бы в современном мире вы не получили хотя бы одного подобного письма. В 2021 году пользователи Microsoft 365 (<https://oreil.ly/YDTnE>) подверглись такой массовой атаке: они получили электронное письмо с вложением, которое, как описывалось в тексте письма, содержало подробную информацию об изменении цен, но при открытии вложения запускался сценарий, фиксирующий учетные данные пользователей.

Межсайтовый скриптинг

При атаке с помощью межсайтового скриптинга (cross-site scripting, XSS) злоумышленники используют незащищенный веб-сайт и внедряют код, манипулирующий поведением приложения. Например, они могут попытаться внедрить код, который позволит им пересылать платежные реквизиты клиентов на собственные серверы. В 2018 году компания British Airways (<https://oreil.ly/OuPZU>) стала жертвой XSS-атаки, в результате которой были раскрыты данные кредитных карт 380 000 клиентов. Компания была оштрафована на крупную сумму из-за отсутствия надлежащей защиты.

Программы-вымогатели

Программы-вымогатели — это вредоносное ПО, которое блокирует систему до тех пор, пока злоумышленнику не будет выплачен выкуп. В 2019 году канал Weather Channel (<https://oreil.ly/SsDIS>) отключился на час из-за атаки такой вредоносной программы-вымогателя на их сеть. Поскольку у компании были резервные серверы, ей удалось успешно преодолеть этот эпизод.

Подделка cookie-файлов

Подделка cookie-файлов — это метод манипулирования cookie-файлами, хранящими информацию о пользователях на сайте и помогающими получить доступ к учетным записям пользователей. В 2017 году Yahoo! сообщила, что около 32 млн

учетных записей пользователей (<https://oreil.ly/6natu>) были взломаны после того, как хакеры получили доступ к собственному коду компании и научились подделывать cookie-файлы, что позволило им получить доступ к учетным записям.

Криптоджекинг

В наши дни широко распространенной атакой стал криптоджекинг (cryptojacking). Это деятельность по тайному майнингу криптовалют с использованием устройств других людей без их разрешения. Обычно боты сканируют общедоступные репозитории GitHub для поиска ключей доступа к инфраструктуре (например, AWS). Обнаруженные таким способом экземпляры эксплуатируются в течение нескольких секунд, что приводит к огромным потерям для владельцев инфраструктуры. В 2018 году жертвой незаконного майнинга стала компания Tesla Inc. (<https://oreil.ly/f4H9L>).

Атаки, обсуждаемые в этом разделе, — это лишь верхушка айсберга. Как показано на рис. 7.2, на разных уровнях — приложения, инфраструктуры и сети — может происходить гораздо больше типов атак. Каждый день хакеры продолжают придумывать новые их виды. Главная обязанность команды разработчиков ПО — оставаться на шаг впереди и защищать своих клиентов и бизнес. Часто ее неисполнение грозит наступлением юридической ответственности, поскольку правительства разработали такие правила, как Общий регламент ЕС по защите данных (European Union's General Data Protection Regulation, GDPR) и Пересмотренная директива о платежных услугах (Revised Payment Services Directive, PSD2), для обеспечения безопасности конфиденциальных данных.



Рис. 7.2. Основные угрозы для программных систем

Модель угроз STRIDE

Примеры из реальной жизни, которые мы обсуждали в предыдущем разделе, ясно иллюстрируют мотивы злоумышленников: они стремятся захватить данные пользователей или компаний, финансы, инфраструктуру или злоупотребить ими, помешать получать доступ к услугам, испортить репутацию бренда. Это основные активы, которые мы должны защищать в любом приложении, и важно правильно оценивать масштаб проблемы. Однажды я работала над проектом центральной системы безопасности для банка. Количество потенциальных угроз безопасности, которые мы рассмотрели и которые могут привести к компрометации любого из этих пяти активов, было поистине поразительным.

Аналогично, размышляя о безопасности своего приложения, вы должны подумать обо всех возможных рисках, которые могут поставить под угрозу активы приложения. Для их идентификации можно использовать модель угроз STRIDE, которую придумали Лорен Конфельдер и Праэрит Гарг из Microsoft. STRIDE — это аббревиатура, образованная от названий классов угроз: **S**poofed identity (спуфинг), **T**ampering with inputs (подмена входных данных), **R**epudiation of actions (отказ от действий), **I**nformation disclosure (раскрытие информации), **D**enial of service (отказ в обслуживании) и **E**scalation of privileges (несанкционированное получение прав). Вы можете рассматривать их по одному и обсуждать все возможные угрозы этого вида для вашего приложения.

Давайте поговорим о каждом из этих классов угроз безопасности.

Спуфинг

Спуфинг — это класс атак, выполняя которые хакер выдает себя за другого человека, чтобы получить доступ к активам. Вспомните приведенный ранее пример социальной инженерии, когда ИИ представлялся начальником генерального директора, чтобы убедить его перевести деньги. Кражи личных данных сегодня широко распространены благодаря социальной инженерии, фишингу, внедрению вредоносного ПО и подсматриванию (шпионажу, когда злоумышленник подсматривает при вводе личной информации, такой как пароль или PIN-код банковской карты), и это лишь некоторые из атак.

В число механизмов защиты для противодействия этому типу угроз входят многофакторная аутентификация, рекомендации по выбору надежных паролей и шифрование при сохранении данных и передаче учетных данных.

Подмена входных данных

Подмена входных данных предполагает изменение в приложении чего-либо — кода, данных, адреса в памяти и т. д., чтобы нарушить их целостность. Обычно это делается внедрением вредоносного кода, например сценария, в пользовательский

интерфейс или на другие уровни. Так в упомянутом ранее примере с British Airways хакер изменил поведение сайта, внедрив скрипт для сбора данных кредитных карт клиентов.

В число защитных мер против этой угрозы входят добавление соответствующих проверок (например, проверки полей ввода, чтобы предотвратить отправку SQL-запросов), механизмы аутентификации и авторизации, а также другие стандартные методы обеспечения безопасности¹ во время написания кода, помогающие избежать уязвимостей, которые могут привести к внедрению кода.

Отказ от действий

Под отказом подразумевается такая атака, когда действия злоумышленника не могут быть доказаны или отслежены. Например, клиент может отказаться от получения товара после доставки, если нет подтверждения получения. Это критический аспект, который следует учитывать при разработке функциональности, поскольку он может привести к потере товаров, репутации и денег, а иногда и к судебным искам. Для борьбы с этой угрозой приложение должно иметь адекватные логи и механизмы аудита, гарантирующие невозможность отказа.

Раскрытие информации

Угрозы раскрытия информации связаны с получением неавторизованными лицами доступа к активам приложения. Как мы видели на примере Twitter, сотрудники могли видеть открытые пароли пользователей, к которым у них не должно было быть доступа. В этом случае раскрытие было непреднамеренным следствием проектирования, и, к счастью, никаких нарушений безопасности не произошло. Однако атаки с целью получения несанкционированного доступа к информации — обычное явление. Один из популярных подходов — настроить вредоносное ПО на фоновое прослушивание законного сайта и передачу информации с него хакеру. Это называется атакой «человек посередине». Чтобы обеспечить должную защиту от такого рода угроз, следует встроить в приложение надежный механизм авторизации, шифровать все конфиденциальные данные и использовать безопасные протоколы передачи.

Отказ в обслуживании

Атака типа «отказ в обслуживании» (Denial of Service, DoS) направлена на организацию сбоев в работе сервисов и приложений, что приводит к потере доходов и нанесению ущерба репутации компании. Одно из проявлений этой угрозы — *распределенная атака типа «отказ в обслуживании»* (Distributed Denial of Service,

¹ Отличное введение в принципы и передовые практики разработки защищенного ПО вы найдете в книге Дэниела Деогана, Дэна Берга Джонссона и Дэниела Савано *Secure by Design* (<https://oreil.ly/ezsTI>) (Manning). (Джонсон Д. Б., Деоган Д., Савано Д. Безопасно by design. — СПб.: Питер, 2021.)

DDoS), когда система намеренно перегружается миллионами запросов от нескольких устройств, что приводит к замедлению ее работы и в итоге к сбою.

В число защитных мер против такого рода угроз входят, например, добавление балансировщиков нагрузки, регулирование запросов на каждый IP-адрес, разрешение запросов только с определенных IP-адресов, создание резервных копий системы, автоматическое развертывание новых компьютеров при увеличении нагрузки и настройка систем мониторинга для оповещения о внезапных скачках количества запросов.

Несанкционированное получение прав

Несанкционированное получение прав происходит, когда злонамеренный пользователь получает несанкционированные привилегии, дающие ему более широкий доступ к системе. Представьте себе хакера, который получает права суперадминистратора! На мой взгляд, это наихудшая угроза, с которой приходится иметь дело, потому что она таит в себе всевозможные риски: кражу личных данных, отказ в обслуживании, финансовые потери и т. д. Лучше всего следовать принципу минимальных привилегий, предоставляя пользователям ровно столько привилегий, сколько им необходимо для решения их задач, и не более того. Этот принцип можно применять и в наших отдельных командах, например предоставляя права на коммит кода только разработчикам и давая их другим только при необходимости. Некоторые полезные методы защиты от этой угрозы: частое обновление токенов доступа, заверение транзакций несколькими подписями, хранение секретов в защищенных хранилищах и т. д.

Итак, используйте модель STRIDE для оценки всех возможных угроз безопасности вашего приложения. Очевидно, что придется подумать и о решениях для предотвращения различных типов атак, и о том, как преодолеть последствия атаки, если она произойдет.

Уязвимости приложений

Научившись думать как хакер, мы обсудили некоторые распространенные типы атак, потенциальные активы, которые хакеры могут попытаться похитить, а также модель, которую можно применять для выявления всех возможных угроз вашему приложению. Следующий шаг — исследовать код приложения для нахождения различных уязвимостей, которыми могут воспользоваться угрозы. Понимание этих уязвимостей поможет добавить защитный код и протестировать его.

Внедрение программного или SQL-кода

Злоумышленник может внедрить вредоносные команды или SQL-запросы, чтобы изменить поведение сайта, если он не защищен. В примере 7.1 показан запрос для получения записи об учащемся по имени.

Пример 7.1. SQL-запрос в коде, принимающий входную переменную

// SQL-запрос в коде, получающий запись об учащемся по имени

```
SELECT * FROM Students WHERE name = '$name'
```

Как видите, запрос принимает входную переменную `$name` от пользователя. Он написан таким образом, что если злонамеренный пользователь вместо имени студента введет SQL-запрос, удаляющий всю таблицу, то он выполнится совершенно нормально, как показано в примере 7.2.

Пример 7.2. Внедренный SQL-запрос удалит всю таблицу

// Если злоумышленник введет в пользовательском

// интерфейсе такую строку:

Name: Alice'; DROP TABLE Students; --

// Приложение выполнит такой запрос:

```
SELECT * FROM Students WHERE name = 'Alice'; DROP TABLE Students; --'
```

Поэтому необходимо тестировать проверку правильности ввода.

Межсайтовый скриптинг

Как обсуждалось ранее, межсайтовый скриптинг предполагает выполнение сценария в браузере жертвы, который позволяет злоумышленнику перехватить контроль над сеансом пользователя, перенаправить пользователя на вредоносный сайт или даже изменить код, чтобы нарушить работу веб-сайта. Атаки этого вида оказываются успешными, когда нет проверки или надлежащей санации пользовательского ввода.

Например, пользователь Twitter опубликовал простой сценарий JavaScript (рис. 7.3), чтобы выявить XSS-уязвимость в приложении TweetDeck. Код заставляет твит автоматически ретвититься каждый раз, когда он появляется в чьей-то ленте, после чего выводится всплывающий диалог с предупреждением. Этого можно было бы избежать, если бы приложение правильно проверяло текст сообщения на наличие скрытых в нем сценариев, но, поскольку этого сделано не было, браузеры пользователей благополучно выполнили сценарий.



Рис. 7.3. Твит с XSS

Незакрытые известные уязвимости

Если приложение зависит от стороннего программного обеспечения (например, ОС, библиотек, фреймворков, инструментов), уязвимость в любом из них может быть использована для получения доступа к системе. Такие уязвимости часто обнаруживаются и устраняются, а разработчики рассылают исправления как обновления. Однако команды могут не обновлять регулярно все уязвимые компоненты своих приложений, в результате чего оказываются незащищенными. Инструменты могут оказать некоторую помощь. Например, GitHub Dependabot (<https://oreil.ly/Нро7р>) позволяет автоматически обновлять любые зависимости с известными уязвимостями. Аналогично инструменты сканирования уязвимостей, такие как Snyk и OWASP Dependency-Check, помогают выявить уязвимые компоненты.

Аутентификация и неправильное управление сеансом

Иногда ненадежными оказываются механизмы аутентификации на веб-сайте, оставляя злоумышленникам возможность красть токены сеансов и использовать полученные привилегии. Если вы храните идентификаторы сеансов и конфиденциальные пользовательские данные в cookie-файлах сеанса, то желательно почаще обновлять их и объявлять недействительными старые cookie-файлы. Кроме того, необходимо следить за такими уязвимостями, как раскрытие идентификаторов сеансов в URL, применение соединений без шифрования для отправки конфиденциальных данных аутентификации и т. д.

Незашифрованные личные данные

Пользователи часто становятся жертвами распространенной уязвимости, выражающейся в хранении личных данных в незашифрованном виде, как в описанном ранее случае, когда были похищены и сохранены в базе данных номера телефонов пользователей Facebook. Вы должны гарантировать невозможность появления незашифрованных личных данных в логах, базах данных, репозиториях кода, проектной документации, общедоступных сервисах и т. д. Кроме того, выбирайте высокопроизводительные криптографические алгоритмы (<https://oreil.ly/gWBsM>), такие как AES, HMAC или SHA-256 с динамическими методами модификации (<https://oreil.ly/SKoYx>), для защиты данных при передаче и сохранении¹.

¹ Дополнительную информацию по этой теме вы найдете в книге Уэйда Траппа и Лоуренса К. Вашингтона Introduction to Cryptography with Coding Theory (<https://oreil.ly/BDSmG>), 3-е издание (Pearson).

Неправильная настройка приложения

Часто ошибкой оказывается предоставление полных прав администратора всем, кто пользуется приложением, по единственной причине — чтобы сэкономить на затратах на обслуживание. Неправильная настройка разрешений для пользователей, папок, систем и т. д. может привести к несанкционированному повышению привилегий и доступу к данным приложения, например к базам данных и конечным точкам сервисов администрирования, которыми можно легко злоупотребить. Команды должны строго придерживаться принципа наименьших привилегий, о котором говорилось ранее.

Раскрытие секретов приложения

Распространенной практикой, которая приводит к компрометации, является включение секретных данных, таких как учетные данные среды, учетные данные суперпользователя и т. д., в код и конфигурационные файлы в виде обычного текста. Способом противостояния таким угрозам является использование сервисов управления секретами, таких как защищенные хранилища, и получение секретов только оттуда. Это относится к коду приложения, конвейерам CI/CD, конфигурационным файлам и всем другим местам, где может понадобиться доступ к секретам.

В списке в этом разделе выделен ряд уязвимостей, с которыми обязательно нужно разобраться во время разработки и тестирования. Open Web Application Security Project (OWASP) — некоммерческая организация, возглавляемая сообществом, определила десять наиболее распространенных уязвимостей в Интернете (<https://oreil.ly/uXFbn>), о которых вам может быть интересно прочитать.

Моделирование угроз

Зайдя так далеко, вы, возможно, уже подумали об угрозах и уязвимостях, которым может подвергнуться ваше приложение прямо сейчас. В этом разделе мы обсудим методический подход к моделированию угроз — структурированный способ агрегирования всех потенциальных угроз безопасности, который вы сможете применить к своему приложению.

Обычно рекомендуется моделировать угрозы для каждой небольшой области приложения. Например, вы можете потратить 15 минут на моделирование угроз для каждой пользовательской истории. Смоделировав угрозы, сможете расставить их приоритеты на основе степени воздействия и вероятности риска, а затем включить решения как часть истории или как новую функцию. При определении приоритетов угроз используйте общее эмпирическое правило: *стоимость создания мер защиты для борьбы с потенциальной угрозой не должна превышать стоимости актива, который вы пытаетесь защитить.*

Предположим, ваша команда разрабатывает платформу для блогов. Прежде чем создавать домашнюю страницу, вы тратите 15 минут на моделирование угроз и обнаруживаете потенциальную угрозу, которая может привести к нарушению работы страницы в результате атаки программы-вымогателя. В качестве решения команда предлагает реализовать систему мониторинга безопасности. По их оценкам, она будет стоить 400 тыс. долларов в год. Стоит ли внедрять это решение для защиты от угрозы программ-вымогателей? Не факт, потому что затраты могут оказаться выше годовой прибыли компании. Кроме того, часто ли происходят атаки программ-вымогателей на платформы блогов? Вероятность этого очень мала. В то же время такую угрозу, как атака внедрением кода на веб-сайт электронной коммерции, которая может привести к потере данных кредитных карт, можно считать угрозой с потенциально значительным ущербом и высокой вероятностью.

После того как вы определите угрозы и расставите приоритеты, найдите решения в той же пользовательской истории или при необходимости создайте для этой цели новые истории «нарушителей» или «злонамеренных пользователей», например:

«Как злонамеренный пользователь, я не могу внедрить код для перенаправления содержимого веб-сайта».

На основе моделирования угроз и критериев приемлемости историй злоумышленников вы можете создать тестовые сценарии, связанные с безопасностью, для целей разработки и тестирования.

Этапы моделирования угроз

Давайте подробнее рассмотрим порядок моделирования угроз. Рекомендуется выполнять это упражнение в команде, в которой представлены все роли. Хорошо также применять доску и цветные стикеры — это поможет зафиксировать мысли членов вашей команды и быстро их классифицировать. В распределенных командах для проведения упражнений можно использовать такие инструменты, как MURAL. Как только команда будет собрана, выполните следующие этапы.

Определение функции

На первом шаге определите функцию, для которой моделируются угрозы. Затем выясните возможные сценарии и типы пользователей или участников системы. Как только это станет ясно, отобразите поток данных от одного компонента к другому. Так вы охватите сценарии и типы пользователей, потоки данных и связи между компонентами системы.

Определение активов

Второй шаг — выявите активы функции, которые необходимо защитить. Обсудите последствия потери каждого актива и определите серьезность риска.

Анализ с позиции злоумышленника

Затем встаньте на сторону злоумышленника и начинайте думать как хакер, придумывая способы атаки на активы приложения. Команда должна действовать под лозунгом: «Сломаем систему!» Используйте модель STRIDE, чтобы структурировать обсуждение. Позвольте воображению течь свободно, не вдаваясь в такие подробности, как действительно ли что-то представляет угрозу или нет, и записывайте все идеи на стикерах.

Расставьте приоритеты угроз и соберите истории

Проанализируйте вероятность выявленных вами угроз и потенциальный ущерб от них и расставьте их приоритеты. Запишите их как истории злоумышленников, чтобы команда могла принять меры для борьбы с ними после мозгового штурма по моделированию угроз.

Теперь, познакомившись с основами, вы готовы получить опыт из первых рук, выполнив типовое упражнение по моделированию угроз.

Упражнение по моделированию угроз

Предположим, что у нас есть приложение для управления заказами в розничном магазине (их создания/просмотра/обновления/удаления). Приложение имеет веб-интерфейс и REST-сервисы для выполнения бизнес-операций с заказами, хранящимися в базе данных. Перейдем к первому этапу и определим участников, потоки данных и связи между разными компонентами.



Это упражнение предназначено лишь для ознакомления с этапами моделирования угроз и не определяет точную модель угроз для системы управления заказами.

Пользователи системы:

- продавец-консультант, добавляющий, редактирующий и отменяющий заказы;
- системный администратор, управляющий инфраструктурой, конфигурациями и развертываниями;
- руководитель службы поддержки клиентов, который использует приложение, чтобы отвечать по телефону на вопросы, связанные с состоянием заказов.

Пользовательский сценарий прост: продавец магазина и руководитель службы поддержки должны войти в приложение, чтобы просмотреть список последних заказов, и им предоставляются возможности управления заказами. На рис. 7.4

показаны связи между компонентами и потоки данных между ними, помогающие пользователю преодолеть свой путь.

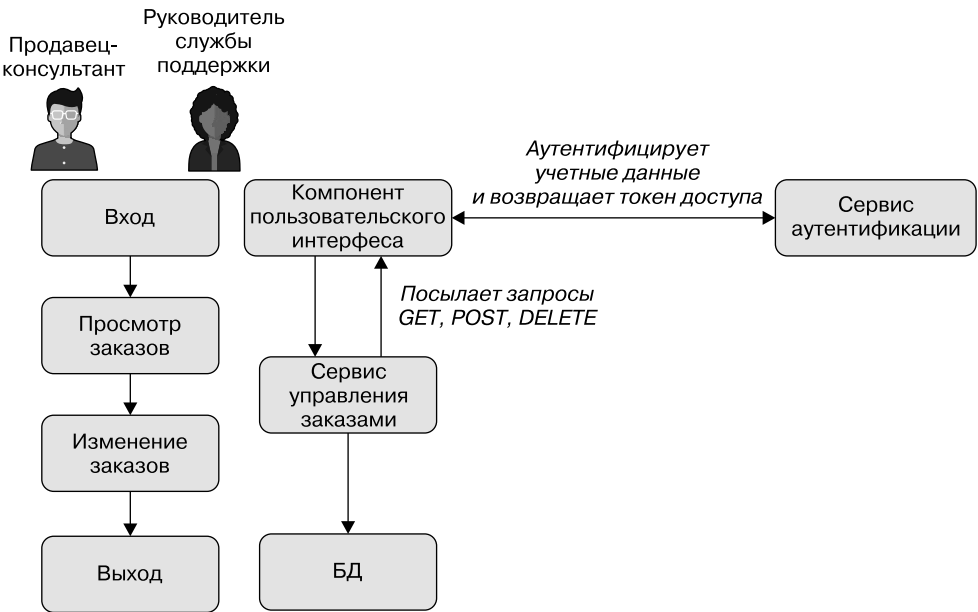


Рис. 7.4. Пользовательские сценарии и потоки данных в системе управления заказами

Аналогично системный администратор должен иметь возможность входить в систему и запускать сценарии или настраивать инфраструктуру (рис. 7.5).



Рис. 7.5. Пользовательский сценарий для администратора

Далее давайте обсудим активы, которые необходимо защитить. Вот некоторые из них.

1. Важнейшим для бизнеса активом является информация о заказе. Клиенты будут недовольны, если их заказы будут изменяться без их ведома, что приведет к потере репутации.
2. Заказы содержат личные данные клиентов, такие как имена, номера телефонов, платежные реквизиты и домашние адреса. Любое раскрытие конфиденциальной

информации приведет к судебным искам и причинит вред клиентам, поэтому данные о клиентах — это еще один важный актив.

3. В базе данных содержится вся информация о продажах магазина. Ее утечка опасна и для клиентов, и для бизнеса, потому что данные могут быть проданы на черном рынке или конкурентам.
4. Инфраструктуру, на основе которой работает приложение, тоже крайне важно защитить, потому что любой ее простой приведет к сбоям в оформлении и исполнении заказов и снижении продаж.

Мы завершили первые два этапа моделирования угроз. Далее нужно проанализировать ситуацию с позиции злоумышленника! Отложите книгу ненадолго. Вспомните, как выглядят пользовательские сценарии и потоки данных, и подумайте, как можно получить контроль над активами. Задействуйте модель STRIDE, чтобы структурировать мышление. Когда закончите, сравните полученную информацию с возможными угрозами, перечисленными на рис. 7.6.

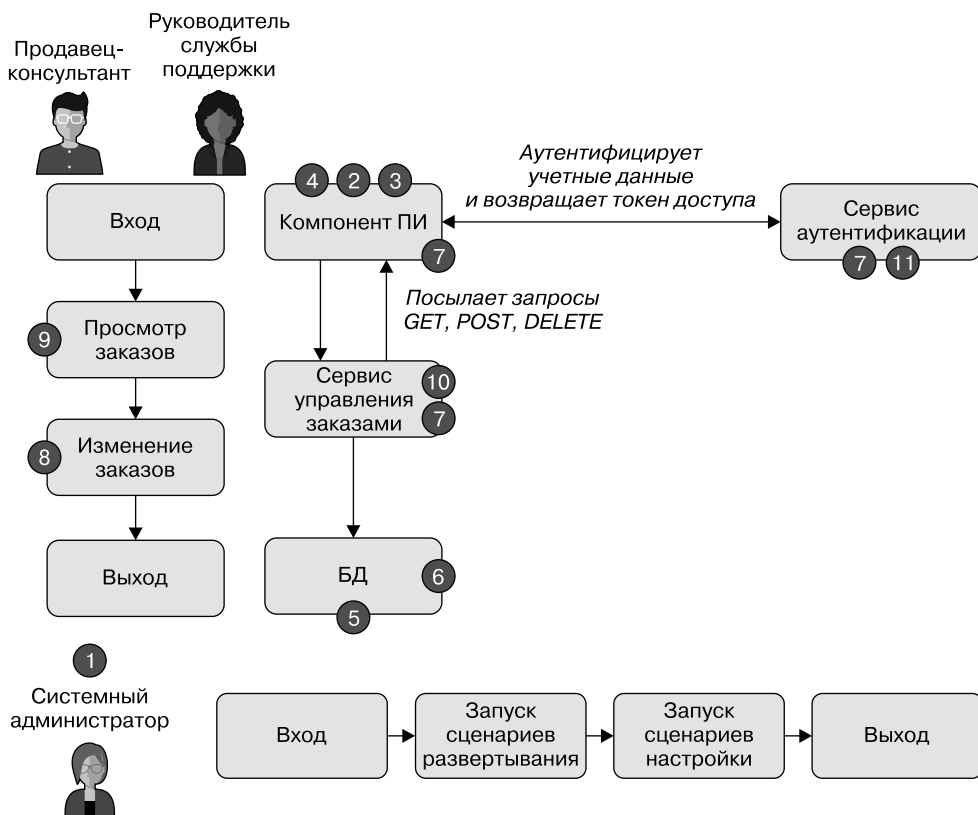


Рис. 7.6. Схема выявленных угроз

Рассмотрим каждый их классов угроз модели STRIDE.

Спуфинг

1. В отношении системного администратора можно прибегнуть к трюкам социальной инженерии, использовать подсматривание или вредоносное ПО, чтобы получить его учетные данные для входа. Поскольку роль системного администратора обладает широчайшими возможностями, злоумышленник может задействовать их для вывода инфраструктуры из строя.
2. Продавец может забыть выйти из системы, и любой присутствующий в магазине сможет использовать активный сеанс, чтобы изменить адреса доставки существующих заказов (например, ввести собственный адрес).

Подмена входных данных

3. Злоумышленник может получить доступ к конечным точкам службы заказов из любого открытого сеанса браузера и подделать заказы, поскольку конечные точки могут быть не защищены.
4. Для кражи платежных данных клиента может использоваться прием инъекции кода при размещении заказа.

Отказ от действий

5. Системный администратор, обнаружив, что его действия не логируются, может создавать объемные заказы для своей семьи и друзей, напрямую вставляя записи в базу данных и запуская соответствующие процессы.

Раскрытие информации

6. Если база данных будет атакована через черный ход, то вся хранящаяся в ней информация окажется раскрытой, потому что данные хранятся в открытом текстовом виде.
7. Кража паролей из незашифрованных логов или других хранилищ позволит злоумышленнику подделывать данные в заказах.
8. Действия руководителя службы поддержки клиентов никак не ограничиваются. Он должен лишь сообщать клиентам текущую информацию о состоянии заказов, но у него тоже есть возможность редактировать заказы. Он может вступить в сговор с злоумышленником и злоупотребить своими разрешениями.
9. Конечная точка `/viewOrders` позволяет получить любое количество записей. Если скомпрометировать эту конечную точку, то ее можно будет использовать для просмотра всех заказов. Нам следует подумать об уменьшении радиуса взрыва.

Отказ в обслуживании

10. Злоумышленник может провести DDoS-атаку и вывести из строя сервис управления заказами, что приведет к снижению продаж.

Несанкционированное получение прав

11. Если злоумышленнику удастся завладеть учетными данными администратора, он сможет добавить новых пользователей или повысить привилегии существующих, чтобы иметь повышенный уровень привилегий доступа к системе в будущем. Он также сможет незаметно создавать, изменять или удалять заказы, поскольку отсутствуют логи, фиксирующие действия системного администратора.

Как видите, даже в небольшой системе с несколькими компонентами и пользователями существует множество точек для атак. Представьте, сколько их будет в реальной системе с тысячами компонентов и пользователей!

Следующий шаг — определение приоритетов угроз и сбор историй. В зависимости от вероятности и серьезности выявленных здесь угроз мы можем добавить новые истории пользователей и злоумышленников, связанные с безопасностью, например такие.

1. «Как злоумышленник, я не должен видеть информацию о клиенте, даже если получу доступ к базе данных».
2. «Как злоумышленник, я не смогу воспользоваться открытыми сеансами в браузере».
3. «Как злоумышленник, если я получу доступ к учетным данным системного администратора или руководителя службы поддержки клиентов, я не смогу редактировать заказы».
4. «Как продавец в магазине, я должен быть единственным человеком, уполномоченным отправлять запросы на редактирование в сервис управления заказами».
5. «Мне, как продавцу в магазине, часто приходится менять пароль для большей надежности».

Как упоминалось ранее, чтобы обнаружить все потенциальные угрозы для приложения, вам следует выполнять упражнения по моделированию угроз итеративно на протяжении всего цикла разработки, сохраняя при этом небольшое количество тестов. Вероятно, вы обнаружите новые угрозы для старых функций, когда станете проводить мозговой штурм, чтобы выявить угрозы для новых функций.

Тестовые сценарии защиты по результатам моделирования угроз

Построив модель угроз, вы сможете понять, какими способами возможно атаковать ваше приложение. А определив истории злоумышленников, получите яркое представление о тестовых примерах, связанных с безопасностью. Следующий шаг после моделирования угроз — применение метода исследовательского тестирования, описанного в главе 2, и сбора тестовых сценариев безопасности для каждого уровня приложения.



Нулевое доверие — это принцип, который предписывает не доверять никому, будь то человек или компонент, даже если он находится внутри периметра защиты. Архитектуры с нулевым доверием проверяют подлинность каждого запроса перед его выполнением, задействуя протокол аутентификации и авторизации, такой как OAuth 2.0, проверяющий подлинность с применением токенов на предъявителя (<https://oreil.ly/RLmbH>). Далее вы увидите использование этих токенов в тестовых примерах.

Вот несколько тестовых сценариев для проверки безопасности нашего сервиса управления заказами на разных уровнях, придуманных, исходя из предположения, что архитектура нулевого доверия реализована с помощью OAuth 2.0.

1. На уровне пользовательского интерфейса убедиться:

- что по истечении времени сеанса пользователю будет предложено снова выполнить вход;
- что учетные данные пользователя блокируются после заданного количества неудачных попыток входа;
- что поля ввода проверяются на наличие недопустимых входных данных, например кода JavaScript, запросов SQL и т. д.;
- что токены доступа действуют в течение короткого времени. При этом пользовательский интерфейс должен автоматически запрашивать токен обновления, чтобы пользователь оставался в системе до истечения сеанса;
- что системный администратор и руководитель службы поддержки клиентов не имеют возможности редактировать заказы в пользовательском интерфейсе.

2. На уровне API убедиться:

- что повторное применение токена доступа с истекшим сроком действия приводит к возврату ответа 401 Unauthorized (хотя 400 предпочтительнее, чтобы избежать раскрытия дополнительной информации);

- что значения параметров API проверяются на наличие недопустимых данных (подобно полям ввода в пользовательском интерфейсе) и, если проверки не пройдены, API возвращают ошибку 404;
 - что конечная точка `/editOrder` возвращает ответ 401 Unauthorized, если используется токен доступа системного администратора или руководителя службы поддержки клиентов.
3. На уровне БД убедиться:
- что пароли хранятся в БД в виде хешей (с применением динамических модификаций) согласно рекомендациям NIST (<https://oreil.ly/RzkYf>);
 - что конфиденциальные данные клиента хранятся в базе данных в зашифрованном виде.
4. На уровне логов приложения убедиться:
- что пароли не записываются в текстовом виде в логи приложения;
 - что конфиденциальная информация пользователя не записывается в текстовом виде в логи приложения;
 - что в логах приложения фиксируются все действия, выполняемые в системе, в том числе системным администратором, с отметками времени.

Это лишь несколько тестовых сценариев, связанных с безопасностью. Возможно, вы сможете придумать больше! Я надеюсь, что это упражнение помогло вам понять, как команды могут внедрить защиту в жизненный цикл разработки ПО от моделирования угроз во время анализа до разработки решений для предотвращения потенциальных угроз и тестирования аспектов безопасности.

Стратегия тестирования безопасности

Практики, обсуждавшиеся до сих пор, — проведение 15-минутного моделирования угроз для каждой пользовательской истории, написание историй злоумышленников, многоуровневая организация мер защиты, создание тестовых сценариев и т. д. — помогут вам укрепить свои системы. Еще один важный шаг — добавление механизмов, обеспечивающих непрерывную обратную связь о потенциальных уязвимостях в разрабатываемом коде, чтобы иметь возможность устранить их как можно скорее. Здесь вам нужно подумать о тестировании безопасности на ранних этапах разработки. Далее обсудим стратегию тестирования безопасности, реализующую этот подход.

На рис. 7.7 показана стратегия тестирования безопасности на ранних и различных этапах к промышленной эксплуатации, начиная с разработки.

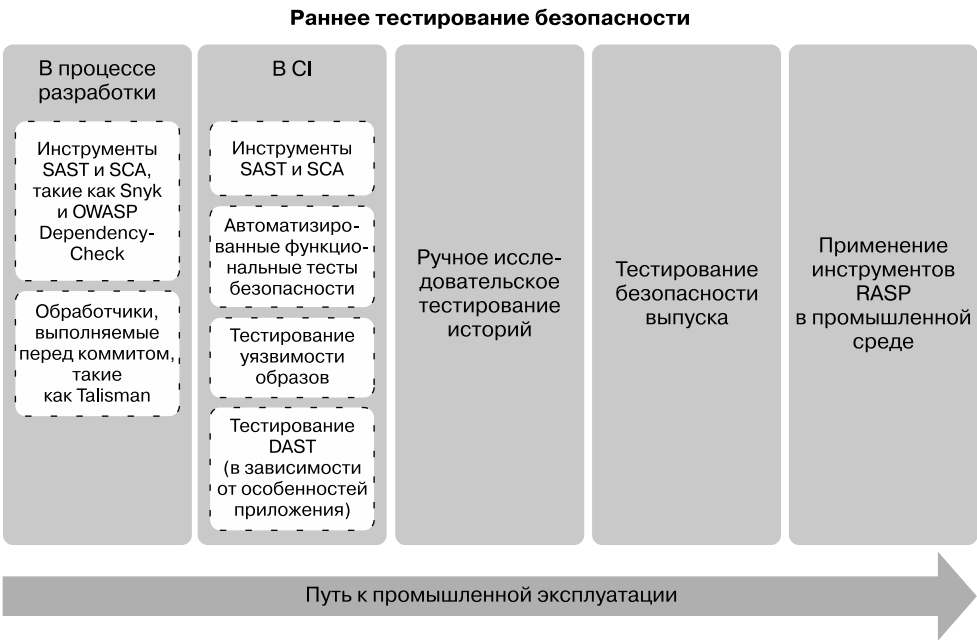


Рис. 7.7. Стратегия тестирования безопасности на ранних этапах разработки ПО

Рассмотрим инструменты и методы, которые вы можете использовать на разных этапах.

Статическое тестирование безопасности приложений (Static Application Security Testing, SAST)

SAST — это метод статического анализа исходного кода, байт-кода и скомпилированного кода приложения на наличие известных уязвимостей. Например, один из вариантов — сканирование кода приложения на наличие незашифрованных конфиденциальных данных. Инструменты SAST имеют разную форму: плагины, библиотеки и решения SaaS (такие как плагин Snyk IDE и Checkmarx SAST, сканирующие код) — и могут интегрироваться в конвейеры CI для проверки каждого коммита. Статическое тестирование безопасности — важная часть идеологии раннего тестирования, потому что помогает обнаруживать проблемы во время разработки.

Talisman, хотя это и не совсем инструмент SAST, специально разработан для сканирования кода приложения для выявления в нем секретов, таких как закрытые ключи, учетные данные среды и т. д., и его можно интегрировать как обработчик, выполняемый перед коммитом. Это поможет предотвратить попадание конфиденциальной информации в репозиторий. Далее в этой главе мы кратко рассмотрим плагин Snyk JetBrains IDE и Talisman.

Анализ составных частей (Software Composition Analysis, SCA)

SCA — это метод выявления уязвимостей в сторонних зависимостях приложения. Инструменты SCA (например, OWASP Dependency-Check и Snyk) особенно полезны тем, кто использует много библиотек с открытым исходным кодом. Они также дают обратную связь во время разработки и могут интегрироваться в CI для проверки каждого коммита. SCA в сочетании с SAST поможет выявить уязвимости уже на этапе разработки. В следующем разделе вы найдете пошаговое упражнение с использованием OWASP Dependency-Check.

Автоматизация функционального тестирования безопасности

Чтобы охватить сценарии функционального тестирования безопасности, можно добавить автоматизированные тесты с помощью инструментов автоматизации функционального тестирования, как описано в главе 3. Например, в упражнении моделирования угроз для сервиса управления заказами, приведенном ранее, мы выявили тестовый сценарий, предназначенный для того, чтобы убедиться: только продавец магазина имеет право редактировать заказы. Этот тест может быть добавлен в качестве теста для проверки сервиса управления заказами.

Сканирование образов

Контейнеры получили широкое распространение как способ упаковки и развертывания приложений. Тестирование уязвимостей в образах контейнеров, если вы их используете, имеет решающее значение. Для этого можно задействовать такие инструменты, как Snyk Container, Anchore и др., которые к тому же прекрасно интегрируются в конвейеры CI. В Docker есть встроенная команда `docker scan`, сканирующая уязвимости в образах Docker. Аналогичную возможность сканирования образов, помещенных в реестр, предлагает Amazon Elastic Container Registry (ECR). При написании инфраструктуры как кода (например, с помощью Terraform или Kubernetes) такие инструменты, как Snyk IaC и `terraform-compliance`, тоже можно использовать для реализации передовых методов обеспечения безопасности.

Динамическое тестирование безопасности приложений (Dynamic Application Security Testing, DAST)

DAST — это метод тестирования черного ящика. Он обнаруживает проблемы безопасности, анализируя ответы приложения на специально сконструированные запросы, имитирующие реальные атаки. Например, инструменты DAST, такие как OWASP ZAP и Burp Suite, пытаются внедрить в приложение вредоносные сценарии, чтобы проверить наличие уязвимостей внедрения кода. Они тоже интегрируются в конвейеры CI, но в зависимости от особенностей приложения могут потребовать довольно много времени для выполнения, поэтому для их запуска выбирайте наиболее подходящий этап CI (см. главу 4).

Подробное упражнение по использованию DAST в лице OWASP ZAP представлено в следующем разделе.



Недавно был разработан новый метод — интерактивное тестирование защищенности приложений (Interactive Application Security Testing, IAST) (<https://oreil.ly/mbpNW>). Его целью является объединение SAST и DAST для анализа поведения приложения во время выполнения. Он основан на инструментовке программного обеспечения и сканирует уязвимости в режиме реального времени. Это новое, бурно развивающееся направление. Примерами инструментов IAST могут служить Contrast Security и Acunetix.

Ручное исследовательское тестирование

В ходе ручного исследовательского тестирования на основе упражнений по моделированию угроз на всех уровнях (пользовательский интерфейс, службы, БД) можно выявить тестовые сценарии, связанные с безопасностью. Как будет показано далее в этой главе, Chrome DevTools и Postman предоставляют множество возможностей для тестирования безопасности.

Тестирование на проникновение (ручное)

В зависимости от критичности приложения и компетентности команды разработчиков ближе к концу цикла поставки можно привлечь профессионального тестировщика безопасности, чтобы убедиться, что приложение не страдает от проблем с безопасностью.

Самозащита приложений во время выполнения (Runtime Application Self Protection, RASP)

Такие методы, как SAST и DAST, обсуждавшиеся ранее, помогают находить уязвимости в коде приложения. Но вам необходим также уровень защиты от атак, выполняемых в промышленной среде. RASP (<https://oreil.ly/29Cw5>) — это метод обеспечения безопасности, включающий в себя мониторинг приложения на предмет потенциальных атак в промышленной среде и их предотвращение. Инструменты RASP (например, Twistlock, Aqua Security) расширяют традиционную идею межсетевого экрана, работая в среде выполнения приложения и накапливая знания о том, какое поведение приложения ожидаемое, а какое — нет. Затем они прослушивают процессы приложения во время выполнения и автоматически принимают меры защиты, такие как автоматическое завершение процессов майнинга криптовалют, проверка данных во входящих запросах и их отклонение, если они оказываются вредоносными, а также помощь в предотвращении атак вредоносных программ (<https://oreil.ly/DDTmj>). Инструменты RASP в настоящее время доступны только на коммерческой основе.

В следующих разделах вы увидите, как некоторые из этих инструментов можно применять на практике.

Упражнения

Приведенные здесь упражнения помогут вам выполнить автоматический анализ составных частей (SCA) с использованием OWASP Dependency-Check и динамическое тестирование безопасности (DAST) с помощью OWASP ZAP, а также интегрировать их в CI для получения постоянной обратной связи.

OWASP Dependency-Check

Как рассказывалось ранее, одна из распространенных угроз — зависимости с уязвимостями. OWASP Dependency-Check — это инструмент SCA с открытым исходным кодом, который сканирует известные уязвимости в библиотеках проекта и внешних зависимостях. Его можно использовать из командной строки или как плагин Jenkins или Maven, есть и другие варианты.

Настройка и рабочий процесс

Выполните следующие шаги, чтобы установить и настроить инструмент проверки зависимостей и запустить сканирование проекта автоматизации тестирования на основе Selenium WebDriver, который мы создали в главе 3.

1. Чтобы установить Dependency-Check в macOS, используйте команду:

```
$ brew install dependency-check
```

Дистрибутив инструмента для других ОС можно скачать с официального сайта (<https://oreil.ly/ICEKY>).

2. После установки запустите сканирование проекта автоматизации тестирования на основе Selenium WebDriver, используя показанную далее команду:

```
// В macOS
```

```
$ dependency-check --project project_name -s project_path --prettyPrint
```

```
// В Windows (файл dependency-check.bat находится в подпапке bin  
// в папке, куда был распакован ZIP-архив  
// с инструментом на предыдущем шаге)
```

```
> dependency-check.bat --project "project_name" --scan "project_path"
```

Эту команду можно интегрировать в конвейер CI, чтобы вызвать сбой в случае обнаружения уязвимостей.

3. Команда создаст в той же папке отчет в формате HTML со списком всех найденных уязвимостей. Проект Selenium WebDriver может иметь или не иметь уязвимости. Пример отчета с уязвимостями для иллюстрации представлен на рис. 7.8.

Summary

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
jquery-1.8.2.min.js		pkg:javascript/jquery@1.8.2.min	MEDIUM	5		3

Published Vulnerabilities

CVE-2012-6708

suppress

jQuery before 1.9.0 is vulnerable to Cross-site Scripting (XSS) attacks. The jQuery() function does not differentiate selectors from HTML in a reliable fashion. In vulnerable versions, jQuery determined whether the input was HTML by looking for the "<" character anywhere in the string, giving attackers more flexibility when attempting to construct a malicious payload. In fixed versions, jQuery only deems the input to be HTML if it explicitly starts with the "<" character, limiting exploitability only to attackers who can control the beginning of a string, which is far less common.

CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Рис. 7.8. Результаты сканирования инструментом OWASP Dependency-Check

Как видите, инструмент сообщил об уязвимости в библиотеке `jquery-1.8.2`. Он ссылается на опубликованную уязвимость `CVE-2012-6708` и объясняет, что «jQuery до версии 1.9.0 уязвим для атак типа межсайтовый скриптинг (XSS)», побуждая нас обновить библиотеку.

OWASP ZAP

OWASP Zed Attack Proxy (ZAP) — это инструмент с открытым исходным кодом, который выполняет динамическое тестирование безопасности приложений (DAST). Он использует предварительно настроенные автоматизированные сценарии, имитирующие атаки на приложение, и с их помощью выявляет набор известных уязвимостей. ZAP, по сути, действует как посредник между браузером и приложением. Он просматривает сообщения, которыми они обмениваются, пытаясь обнаружить признаки известных уязвимостей, и модифицирует их для имитации различных атак. Это позволяет командам, плохо знакомым с основами безопасности, легко находить проблемы в системе защиты. Кроме того, ZAP имеет обширный список конфигураций и дополнений для поддержки множества функций, что позволяет специалистам по безопасности добавлять дополнительные сценарии. Для изучения этих вариантов есть хорошая документация (<https://oreil.ly/v7gmD>). ZAP можно интегрировать с другими инструментами, такими как Selenium, что также упрощает работу в CI.

Давайте познакомимся с ZAP.

Установка и настройка

Чтобы установить ZAP на macOS, примените следующую команду:

```
$ brew install cask owasp-zap
```

Для других ОС используйте установочные файлы с официального сайта (<https://oreil.ly/IXZ9t>).

Рабочий процесс

После установки можно открыть пользовательский интерфейс ZAP Desktop, который выглядит, как показано на рис. 7.9. (На Mac вам, возможно, придется дать разрешение на открытие приложения, потому что его нет в App Store.)

Первый шаг — настройка в ZAP внутренних URL приложений и компонентов пользовательского интерфейса, чтобы он мог имитировать атаки против них. Это можно сделать двумя способами: используя кнопку Manual Explore (Исследование вручную) (см. рис. 7.9) или с помощью ZAP Spider.



Рис. 7.9. Пользовательский интерфейс ZAP Desktop

Исследование вручную. Чтобы вручную исследовать приложение, щелкните на кнопке Manual Explore (Исследование вручную) в пользовательском интерфейсе ZAP Desktop. После этого откроется диалоговое окно (рис. 7.10), предлагающее ввести URL приложения.



Не используйте этот инструмент для тестирования общедоступных веб-сайтов. Тестирование безопасности веб-сайта без разрешения его владельца незаконно.

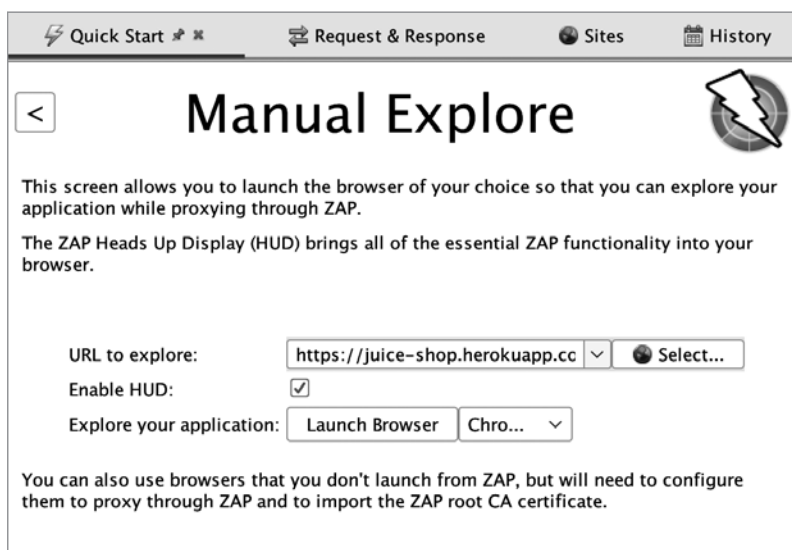


Рис. 7.10. Исследование вручную в ZAP

Для экспериментов проект OWASP предоставляет подопытный веб-сайт OWASP Juice Shop (<https://oreil.ly/BdI8D>), на примере которого можно опробовать приемы тестирования безопасности. Я применила этот URL. ZAP позволяет проводить исследования как из Firefox, так и из Chrome. Как только браузер откроет приложение, пройдите пользовательский сценарий один раз. ZAP просканирует приложение в фоновом режиме и запишет соответствующие детали.

Обратите внимание на флажок **Enable HUD** (Включить проецирование), показанный на рис. 7.10. В режиме проецирования (Heads Up Display, HUD) результаты сканирования отображаются поверх интерфейса приложения в браузере. Это помогает избежать необходимости переключаться между пользовательским интерфейсом ZAP Desktop и браузером во время атак. Если вы включили проецирование, то увидите результаты, отображаемые поверх страницы веб-сайта Juice Shop (рис. 7.11, панели слева и справа).

Исследуя сайт вручную, вы увидите, что значок **Sites** (Сайты) на правой панели отображает дерево сайта, а на вкладке **History** (История) внизу перечислены посещавшиеся URL. ZAP использует эти данные позже, в ходе активных попыток атаковать приложение.

ZAP Spider. ZAP Spider избавляет от бремени ручного исследования веб-сайта. Он автоматически сканирует сайт с помощью Selenium WebDriver и собирает все URL приложений и компонентов пользовательского интерфейса. Простой Spider (серый значок в разделе **Sites** (Сайты) справа) может не иметь возможности перемещаться по компонентам JavaScript, но AJAX Spider (<https://oreil.ly/oifnD>)

(красный значок) способен на это. Вы можете задействовать оба варианта, чтобы полностью изучить приложение.

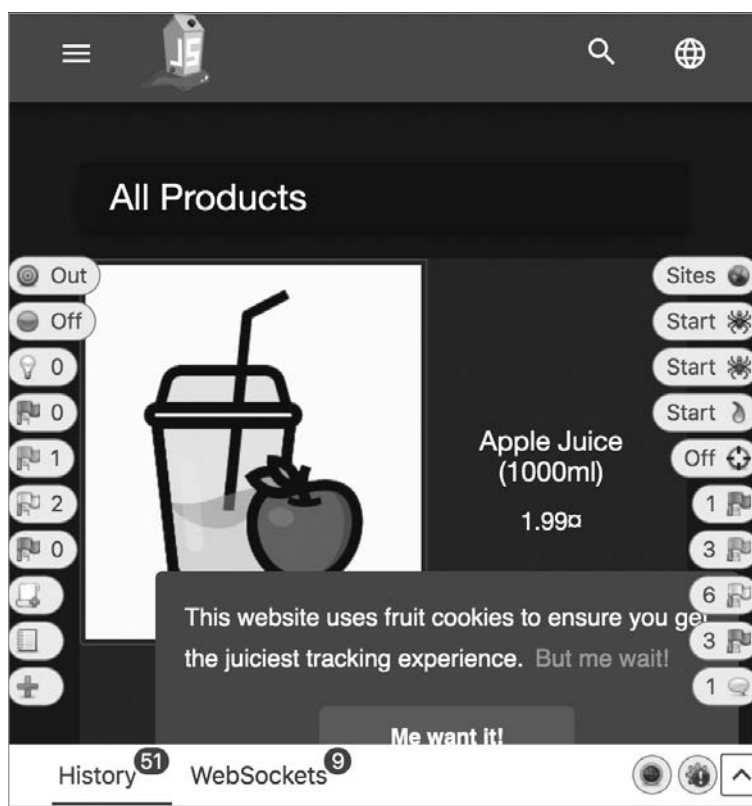


Рис. 7.11. Отображение результатов сканирования поверх интерфейса веб-сайта Juice Shop

Сканирование. Когда ZAP Spider перемещается по приложениям, собирая информацию, он выполняет пассивное сканирование в фоновом режиме. Поддерживает он и другой режим сканирования, называемый активным сканированием, в ходе которого атакует приложение! Попробуйте эти режимы прямо сейчас.

- Пассивное сканирование предполагает чтение сообщений, которыми обмениваются браузер и веб-приложение, и их проверку на наличие уязвимостей. Сами сообщения при этом не изменяются (то есть не подвергаются атаке). Это сканирование выполняется автоматически, поэтому, когда ZAP Spider занимается сканированием, вы увидите предупреждения, отображаемые на правой и левой панелях. Предупреждения распределяются по приоритету в зависимости от серьезности (высокая, средняя и низкая) и группируются под красными, оранжевыми и желтыми флажками соответственно (см. рис. 7.11). Щелкая на этих

флажках, можно вывести подробную информацию о найденных уязвимостях. Кроме того, ZAP Desktop создает подробные логи. Например, в ходе пассивного сканирования ZAP обнаружил на веб-сайте Juice Shop частный IP-адрес (рис. 7.12).

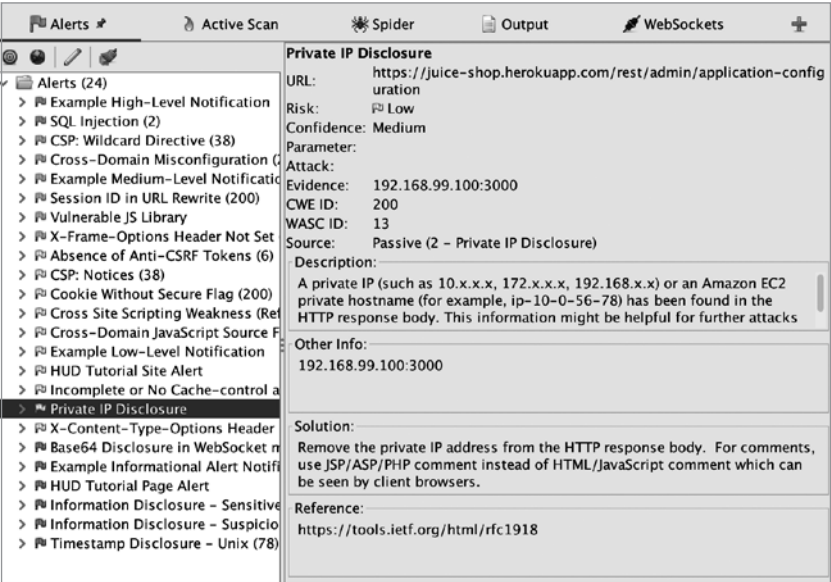


Рис. 7.12. Результаты пассивного сканирования в ZAP Desktop

- Во время активного сканирования ZAP атакует приложение, перехватывая запросы, изменяя их и пересылая туда-обратно, проверяя наличие известных уязвимостей, таких как внедрение SQL-кода и многие другие. Щелкните на четвертом сверху значке на правой панели (под красным пауком), чтобы запустить активное сканирование. Вы сразу увидите, как ZAP перемещается по сайту страница за страницей, имитируя различные атаки. Сканирование занимает некоторое время. По завершении вы увидите найденные уязвимости под цветными флажками. На рис. 7.13 показана уязвимость, обнаруженная на веб-сайте Juice Shop, которая допускает внедрение SQL-кода.

Вот так просто ZAP динамически тестирует безопасность приложений: просто откройте приложение в пользовательском интерфейсе ZAP, щелкайте на значках с изображениями пауков и запускайте пассивное и/или активное сканирование!

Интеграция ZAP в CI. После того как ZAP предоставит список всех уязвимостей приложения, вы должны проанализировать их и решить, какие из них исправить. Это требует времени и опыта. Желательно выполнять это упражнение постоянно, не оставляя работу на потом. Это прекрасный повод интегрировать инструмент в конвейер CI для получения постоянной обратной связи!

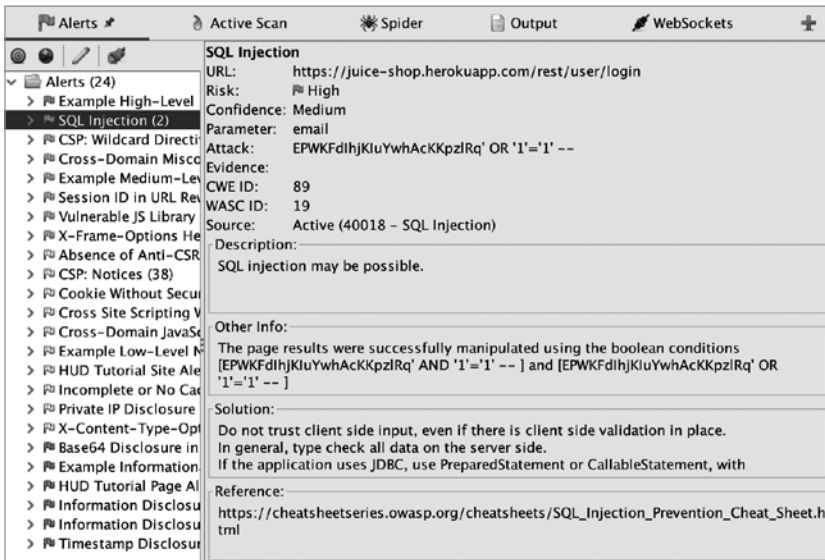


Рис. 7.13. В ходе активного сканирования обнаружена уязвимость, допускающая внедрение SQL-кода



Как упоминалось ранее, активное сканирование может занять много времени, что зависит от сложности приложения, — иногда даже несколько часов. В таких случаях можно интегрировать инструмент в конвейер CI и запускать его в ночном этапе регрессионного тестирования или включить ручной триггер для тестирования каждой пользовательской истории.

Для интеграции с CI ZAP предоставляет следующие API:

- `zap.urlopen(target)` — открывает приложение;
- `zap.spider.scan(target)` — запускает ZAP Spider и выполняет пассивное сканирование;
- `zap.ascan.scan(target)` — запускает активное сканирование;
- `zap.core.alerts()` — выводит результаты сканирования.

Вы можете использовать эти API в простом сценарии JavaScript или Python для запуска сканирования и интеграции с CI с помощью ZAP CLI (<https://oreil.ly/3S67c>).

Также есть возможность встроить ZAP API в функциональные тесты Selenium WebDriver и запускать их как обычные функциональные тесты в конвейере CI. Кроме того, WebDriver может помочь ZAP выполнить вход на веб-сайт, чего он, возможно, не сможет сделать самостоятельно. В примере 7.3 показан пример теста, который сканирует приложение и завершается неудачей, если обнаруживает уязвимости. Обратите внимание на то, что в зависимости от времени сканирования приложения следует добавить соответствующие задержки.

Пример 7.3. ZAP-сканирование в тестах Selenium

```
@Test
public void testSecurityVulnerabilities() throws Exception {

    zapScanner = new ZAPProxyScanner(ZAP_PROXYHOST, ZAP_PROXYPORT, ZAP_APIKEY);
    login.loginAsUser();

    // Шаг 1 – сканирование с использованием ZAP API
    zapSpider.spider(BASE_URL)

    // Шаг 2 – разрешить пассивное сканирование
    zapScanner.setEnablePassiveScan(true);

    // Шаг 3 – запустить активное сканирование.
    // Добавьте вызовы методов задержки.
    zapScanner.scan(BASE_URL);

    // Шаг 4 – записать предупреждения в лог и проверить их количество
    List<Alert> alerts = filterAlerts(zapScanner.getAlerts());
    logAlerts(alerts);
    assertThat(alerts.size(), equalTo(0));
}
```

ZAP создает HTML-отчеты с описанием уязвимостей (рис. 7.14). Их можно сохранять в виде выходных артефактов в CI.

High (Medium)	SQL Injection
Description	SQL injection may be possible.
URL	https://juice-shop.herokuapp.com/rest/user/login
Method	POST
Parameter	email
Attack	EPWKFdIhjKluYwhAckKpzIRq' OR '1'='1' --
URL	https://juice-shop.herokuapp.com/rest/user/login
Method	POST
Parameter	email
Attack	VqqxCXFFxHhqCixYYvCGioKa' OR '1'='1' --
Instances	2
	Do not trust client side input, even if there is client side validation in place. In general, type check all data on the server side. If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'

Рис. 7.14. HTML-отчет, сгенерированный инструментом ZAP

Наконец, если для организации своего конвейера CI/CD вы используете GitHub Actions, то в вашем распоряжении имеется простой вариант интеграции ZAP с применением predefined сценариев базового сканирования OWASP ZAP Baseline Scan (<https://oreil.ly/Ht7hI>) и полного сканирования OWASP ZAP Full

Scan (<https://oreil.ly/aaxT2>) из Actions. Как следует из их названий, они выполняют сканирование с помощью ZAP и добавляют описание проблем в GitHub.

ZAP имеет множество других полезных функций, помимо тех, что мы обсудили, которые позволяют проводить различные виды исследовательского тестирования безопасности приложения. Вот некоторые из них:

- ZAP может использовать спецификации OpenAPI для тестирования безопасности API;
- функция Breaks поможет вставлять в запросы определенные тестовые данные и наблюдать за поведением приложения. Например, с помощью функции Break можно убедиться, что API проверяет входные параметры на предмет внедрения SQL-кода;
- позволяет воспроизвести запрос в браузере;
- дает возможность выделить определенные скрытые ключевые слова в HTML;
- имеет функцию раскрытия всех скрытых полей ввода в приложении;
- предлагает дополнительные готовые сценарии, написанные экспертами, которые при необходимости можно использовать для имитации определенных типов атак.

В целом, ZAP — отличный инструмент, способный научить вас многому в области защиты.

Дополнительные инструменты тестирования

Обсудим еще несколько инструментов, помогающих при статическом и ручном исследовательском тестировании безопасности приложений, чтобы дать вам более широкий обзор инструментов, связанных с безопасностью, которые можно использовать во время цикла поставки программного обеспечения.

Плагин Snyk IDE

Плагин Snyk JetBrains IDE (<https://oreil.ly/8Vq7c>) сочетает в себе возможности SCA и SAST. Он доступен бесплатно и может применяться с любой из JetBrains IDE (IntelliJ IDEA, WebStorm, PyCharm и т. д.). Самое большое его преимущество — он близок к этапу разработки и способствует раннему тестированию. Вы можете запустить сканирование кода приложения и его зависимостей для проверки наличия уязвимостей прямо во время разработки. На рис. 7.15 показаны результаты такого сканирования, отображаемые на нижней панели IntelliJ IDE. Как видите, Snyk выявил уязвимость Information disclosure («Раскрытие информации») в коде приложения. Он также показывает варианты исправления обнаруженных уязвимостей, что упрощает для разработчиков укрепление системы защиты.

Snyk доступен и в виде инструмента командной строки, но только с возможностями SCA, которые необходимо интегрировать в CI. Компания-разработчик предоставляет также набор коммерческих услуг, связанных с безопасностью.

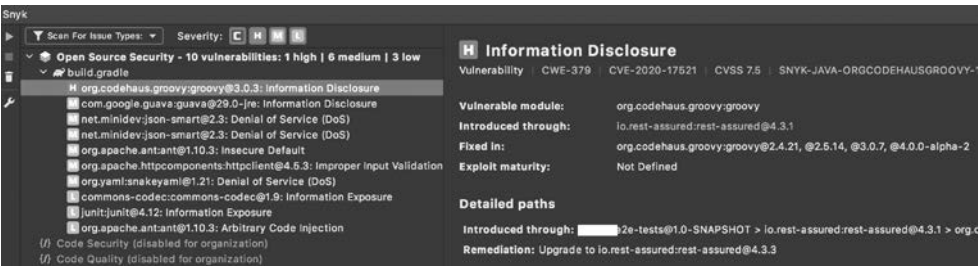


Рис. 7.15. Пример результатов сканирования, полученных плагином Snyk IDE

Talisman, обработка события коммита

Talisman (<https://github.com/thoughtworks/talisman>) — это инструмент с открытым исходным кодом, который сканирует код приложения на наличие в нем конфиденциальной информации, например паролей, ключей SSH, токенов и т. д. Он запускается в тот момент, когда вы предпринимаете попытку отправить данные в систему управления версиями, и сообщает о проблемах, если они обнаружатся. Это очень полезно для предотвращения случайного раскрытия секретов командами разработчиков. Talisman можно настроить как обработчик события коммита или события отправки. В примере 7.4 показан пример результата сканирования при попытке зафиксировать код в Git.

Пример 7.4. Результаты сканирования, произведенного инструментом Talisman

```
$ git commit
Talisman Report:
```

FILE	ERRORS
sampleCode.pem	The filename "sampleCode.pem" failed checks against the pattern ^.+\.pem\$
sampleCode.pem	Expected file not to contain hex-encoded texts such as: awsSecretKey=c99e0c79ddcf5ddb02f1274db2d973f363f4f553ab1692d8d203b4cc09692f79

В данном случае Talisman обнаружил `awsSecretKey` в коде приложения. Поскольку, как обсуждалось ранее, боты сканируют репозитории GitHub в поисках секретов, это важный инструмент, который стоит того, чтобы вы добавили его в свой арсенал.

Chrome DevTools и Postman

Для ручного исследовательского тестирования безопасности различных тестовых сценариев, выявленных в результате моделирования угроз, очень удобно использовать Chrome DevTools и Postman. Возможности Postman мы подробно обсуждали в главе 2. Позволяет этот инструмент и выполнять исследовательское тестирование безопасности, например настраивать передачу токенов аутентификации в запросах к API (рис. 7.16). Эту возможность можно применять для тестирования таких сценариев, как подделка токенов или использование токенов с истекшим сроком действия.



Рис. 7.16. Настройка использования токенов доступа в Postman

Аналогично вкладка Security (Безопасность) в Chrome DevTools сообщает, правильно обслуживается страница через HTTPS или нет (рис. 7.17). Она также сообщит, если ресурсы со сторонних сайтов будут обрабатываться небезопасно, поскольку это может привести к атакам «человек посередине».

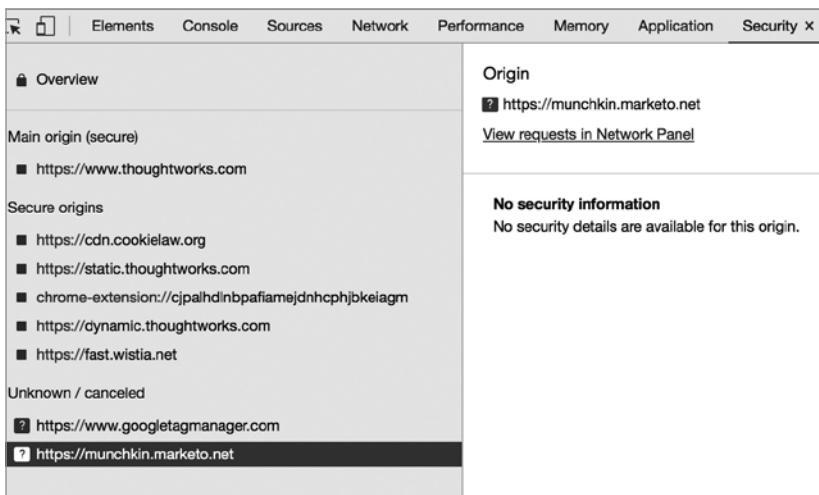


Рис. 7.17. Вкладка Security (Безопасность) в Chrome DevTools

Благодаря инструментам тестирования безопасности, которые мы здесь рассмотрели, у вас должно сложиться четкое понимание того, что тестирование безопасности можно встроить в цикл разработки ПО и за него не обязательно должны отвечать исключительно опытные тестировщики. Избежать серьезных проблем с безопасностью в дальнейшем вашей команде поможет тестирование безопасности на ранних этапах разработки ПО.

Перспективы: защита должна стать привычкой

Мой опыт показывает, что, сколько бы усилий мы ни прикладывали для реализации различных мероприятий, обсуждаемых в этой главе, пытаюсь создать защищенное ПО, если защита не станет привычкой, мы можем оставить в своих приложениях неожиданные слабые звенья, которыми способны воспользоваться недоброжелатели. Фактически некоторые распространенные практики, используемые командами разработчиков ПО, могут легко привести к нарушению безопасности. Например, задумывались ли вы об аспектах безопасности инструментов, которые применяете при разработке и тестировании? Хранит ли какой-нибудь из них свои данные в частном облаке поставщика? Выкладываете ли вы архитектурную схему проекта с излишними деталями на онлайн-портал? Делитесь ли учетными данными в промышленной системе со всеми членами своей команды? Если да, то передавали ли вы их через Slack в обычном текстовом виде? Многие из таких простых действий могут легко привести к компрометации.

Поэтому сделать защиту привычкой — единственный правильный путь. Примерно так же мы оглядываем еду перед тем, как ее съесть, или замечаем, что кто-то следует за нами. Мы выполняем эти проверки неосознанно и естественно. Точно так же мы, как команды разработчиков ПО, должны научиться применять меры защиты бессознательно и естественно. Можно начать так: ежедневно задавать себе вопросы о том, сможет ли какое-либо из простых, непреднамеренных и безвредных действий, которые мы совершаем, привести к нарушению безопасности.

Ключевые выводы

- Киберпреступления в цифровую эпоху распространены гораздо шире, чем вы думаете, а годовой доход киберпреступников, как ожидается, превысит 10 трлн долларов всего за несколько лет.
- Реальные примеры атак показывают, что цифровые платформы взламывают с целью украсть деньги или личные данные, получить доступ к инфраструктуре и т. д. Поэтому очевидно, что безопасность — это не просто хорошая функция.

- Меры безопасности следует встраивать в приложение на всех этапах жизненного цикла его разработки, от анализа до тестирования, чтобы в итоге получить надежную и непроницаемую систему.
- Модель STRIDE обеспечивает структурированный подход к изучению безопасности приложения, который можно применять для моделирования угроз.
- Упражнения по моделированию угроз следует выполнять всей командой для каждой небольшой части функциональности приложения, такой как пользовательская история или функция. Моделирование угроз должно приводить к созданию историй злоумышленников и тестовых сценариев, связанных с безопасностью.
- Разворачивайте стратегии тестирования безопасности на ранних этапах разработки ПО, используя различные инструменты автоматизированного тестирования безопасности (SAST, SCA, DAST и т. д.), а также ручное исследовательское и функциональное автоматизированное тестирование.
- Благодаря наличию доступных инструментов автоматизированного тестирования безопасности командам разработчиков ПО не придется ждать этапа тестирования на проникновение как единственного способа получить обратную связь по проблемам безопасности.
- Самое главное, сделайте защиту своей привычкой.

ГЛАВА 8

Тестирование производительности

Время — деньги!

Бенджамин Франклин

Мы все сталкивались с таким явлением: иногда наши любимые веб-сайты внезапно становились медленными, как ленивцы, и заставляли нас задаваться вопросом: «У меня проблемы с Интернетом?» Приходилось ли вам во время распродажи ждать вечность, пока загрузится веб-сайт? Приходилось ли смотреть на индикатор загрузки и ждать, когда появится сообщение об успешном бронировании билетов на поезд в рождественские каникулы? Или, может быть, приходилось надолго застревать на странице бронирования билетов на блокбастер в кинотеатр? Плохая работа веб-сайта в подобных случаях может вызвать у нас, как клиентов, сильное разочарование.

Если вы хотите уберечь конечных пользователей вашего приложения от подобных разочарований, то придется постоянно оценивать его производительность и работать над ее улучшением. Цель этой главы — познакомить вас со всем, что может пригодиться для измерения или тестирования производительности веб-приложения. В частности, мы рассмотрим такие темы, как ключевые показатели производительности, тестирование производительности API, тестирование производительности фронтенда и раннее тестирование производительности. В упражнениях этой главы у вас также будет возможность попробовать на практике протестировать производительность фронтенда и API.

Тестирование производительности — очень широкая тема. Оно должно проводиться как изнутри, так и снаружи, поэтому по структуре эта глава немного отличается от предыдущих. Здесь вы найдете уже знакомые разделы, но поделенные на две половины.

Начнем с описания всего, что необходимо для быстрого освоения тестирования производительности бэкенда (серверной части), включая упражнения и дополнительные инструменты. После этого переключим внимание на тестирование производительности фронтенда (пользовательского интерфейса). А глобальная стратегия раннего тестирования производительности будет представлена лишь в конце главы.

Введение в тестирование производительности бэкенда

Для начала посмотрим, почему производительность так важна для успеха бизнеса. Здесь я перечислю факторы, влияющие на производительность приложения, ключевые показатели производительности веб-приложения и способы их измерения.

Взаимосвязь производительности, продаж и выходных

В начале главы мы говорили о том, что клиенты разочаровываются из-за низкой производительности приложений. Нам нужно понять, к чему это может привести. Какое значение может иметь задержка в несколько секунд? Существует количественный показатель, который позволяет оценить влияние времени загрузки страницы на поведение клиентов. *Доля отказов* — это процент клиентов, которые покидают веб-сайт после просмотра всего одной страницы.

Среди потенциальных факторов, которые могут увеличить долю отказов, основной вклад вносит производительность веб-сайта. Статистические данные, опубликованные Google (<https://oreil.ly/xQGcV>) и представленные в табл. 8.1, демонстрируют корреляцию между временем загрузки страницы и долей отказов пользователей. Они подтверждают, что с каждой дополнительной секундой задержки бизнес теряет клиентов, уступая их конкурентам.

Таблица 8.1. Статистика Google, связывающая время загрузки страницы и долю отказов

Время загрузки страницы, с	Вероятность отказа увеличивается до, %
1–3	32
1–5	90
1–6	106
1–10	123

И это еще не все: алгоритмы поисковой оптимизации (Search Engine Optimization, SEO) Google присваивают медленным сайтам более низкий ранг, а это означает, что, если ваш сайт недостаточно эффективен, он будет еще глубже погружаться в пропасть! Сама Google стремится к тому, чтобы ее собственный сайт загружался меньше чем за полсекунды (<https://oreil.ly/tULJ9>), и рекомендует ориентироваться на порог 2 с как максимально приемлемое время загрузки веб-сайтов.

Потеря клиентов приводит к снижению продаж, и торговые площадки могут заплатить очень высокую цену за сбой в работе. Например, в 2018 году Amazon

потеряла примерно 72–99 млн долларов (<https://oreil.ly/Q1s5h>), когда ее веб-сайт не смог обслужить весь трафик в день проведения акции Prime Day. Низкая производительность может привести также к потере репутации бренда, особенно в мире, где благодаря социальным сетям негативные отзывы могут распространяться очень быстро.

Небольшое увеличение производительности, напротив, может привести к значительному увеличению продаж. Например, в 2016 году Trainline (<https://oreil.ly/hx3FD>), железнодорожная компания из Великобритании, сократила среднее время загрузки страниц на 0,3 с и ее доход увеличился на 8 млн фунтов (11 млн долларов США) в год. Аналогично поставщик услуг «фронтенд как услуга» (frontend-as-a-service) Mobify (<https://oreil.ly/Lq4Pi>) заметил, что уменьшение времени загрузки его домашней страницы на каждые 100 мс увеличивает количество обращений, что влечет за собой увеличение годового дохода на 380 000 долларов США. Корреляция между продажами и производительностью ясно показывает, что первым шагом к повышению продаж онлайн-бизнеса является анализ производительности его приложений. Это означает, что нам, как командам разработчиков ПО, необходимо создавать и тестировать производительность как можно раньше и чаще, то есть начинать тестирование на ранних этапах разработки ПО.

Один из главных мотивов, побуждающих меня как можно раньше начинать уделять внимание производительности веб-сайта, прост: я люблю выходные и хочу отдыхать в эти дни. Поскольку проблемы с производительностью могут стоить очень дорого, как продемонстрировали предыдущие примеры, и напрямую влияют на репутацию бренда, группы разработчиков ПО обычно испытывают сильное давление, когда от них требуют исправить эти проблемы как можно скорее. Поэтому, если вы не включите тестирование производительности в ранние этапы разработки, то позже вам придется заплатить за это, работая по выходным над тем, чтобы исправить возникающие проблемы с производительностью!

Простые цели производительности

Производительность можно рассматривать как способность приложения одновременно обслуживать большое количество пользователей без значительного ухудшения поведения по сравнению с тем, когда оно обслуживает только одного. То есть производительность не должна падать ниже уровня, приемлемого для конечных пользователей. Итак, чтобы протестировать производительность, сначала необходимо определить ожидаемое количество пользователей в часы пик, а затем убедиться, что на этом уровне нагрузки производительность приложения остается приемлемой.

Что означают слова «приемлемая производительность», во многом зависит от человеческого восприятия. По словам Якоба Нильсена (<https://oreil.ly/OJAUL>), исследователя удобства применения Интернета и компьютера, когда время отклика

сайта составляет менее 0,1 с, пользователь чувствует, что он работает мгновенно. Когда время отклика составляет от 0,2 до 1 с, он ощущает задержку в работе, но сохраняет ощущение управляемости навигации по сайту. Правда, при этом он также чувствует, что пользовательский интерфейс работает медленно, и теряет ощущение потока при выполнении желаемой задачи. Как показывают упомянутые ранее исследования Google, при задержках более 3 с вы рискуете потерять большинство своих клиентов, и она рекомендует поддерживать время загрузки страницы менее 2 с.

Это ваши цели производительности. Чтобы достичь таких хороших результатов, необходимо потратить время на настройку инфраструктуры и оптимизацию, прежде чем запустить свое приложение в эксплуатацию, — это еще одна причина принять стратегию раннего тестирования производительности!

Факторы, влияющие на производительность приложений

Достичь целей в отношении производительности, представленных в предыдущем разделе, совсем не просто, иначе предприятия не теряли бы так много денег из-за проблем с ней. В приложении существует множество факторов, влияющих на его производительность, в том числе перечисленные далее.

Архитектурный дизайн

Архитектурный дизайн играет важную роль в работе веб-сайта. Например, если обязанности веб-сервисов не разделены должным образом, то пользовательскому интерфейсу придется выполнить множество вызовов к разным сервисам, что увеличивает время ответа. Аналогично если механизмы кэширования не настроить должным образом, то это повлияет на производительность веб-сайта.

Выбор технологического стека

Разным уровням приложения требуются разные наборы инструментов. Эти инструменты могут работать несогласованно и ухудшать общую производительность. Приведу лишь один пример: выбор языка программирования (например, Java, Ruby, Go, Python) может заметно повлиять на время холодного запуска AWS Lambda (<https://oreil.ly/UDHFi>).

Сложность кода

Сложный или неоптимальный код (например, сложные алгоритмы, продолжительные операции, отсутствующие или повторяющиеся проверки и т. д.) часто приводит к проблемам с производительностью. Рассмотрим случай, когда поиск выполняется по пустой строке. Конечная точка, отвечающая за поиск, могла бы выполнить простую проверку входных данных и быстро отклонить запрос, а не передавать искомую строку базе данных, только чтобы получить в ответ сообщение об ошибке, и не терять время понапрасну.

Выбор и организация базы данных

Базы данных играют ключевую роль в производительности приложений. Как обсуждалось в главе 5, существуют разные типы БД. Если вашему приложению требуется очень высокая производительность, то выбор подходящего типа базы данных и правильная организация данных внутри нее будут иметь решающее значение. Например, при хранении сведений об одном заказе в нескольких таблицах потребуется выбирать информацию из нескольких таблиц, что приведет к задержке получения окончательного результата. Очень важно правильно структурировать данные с учетом производительности.

Задержка в сети

Сеть является центральной нервной системой любого приложения. Все его компоненты взаимодействуют между собой через какую-то сеть. Поэтому обеспечение хорошей связи между ними имеет решающее значение. Кроме того, конечные пользователи будут взаимодействовать с приложением, применяя собственные сети (2G, 3G, 4G, Wi-Fi). Разработчики ПО не могут повлиять на качество этих сетей, но в силах разработать приложение с учетом слабых сторон сетевых подключений у конечных пользователей. Отказ от задействования в пользовательском интерфейсе тяжелых изображений и передачи больших объемов данных поможет повысить производительность приложения для всех пользователей.

Географическое местоположение приложения и пользователей

Если пользователи вашего сайта находятся в каком-то одном регионе, то физическое размещение сайта рядом с ним уменьшит количество сетевых переходов и, следовательно, задержку. Например, если веб-сайт предназначен для клиентов из Европы, но расположен в Сингапуре, то для подключения к нему сетевым пакетам потребуется выполнить несколько переходов через разные сегменты сети. Но если разместить сайт где-нибудь в Европе, то количество переходов уменьшится, а производительность улучшится. И наоборот, если предполагается, что сайт будет обслуживать клиентов по всему миру, то необходимо разработать стратегию его репликации в разных местах или использования сетей доставки контента (Content Delivery Networks, CDN). Если вы задействуете облачную инфраструктуру, то не забывайте запрашивать экземпляры, которые физически находятся ближе к предполагаемым клиентам. Распространенная ошибка — использование инфраструктуры, расположенной ближе к местоположению команды разработчиков.

Инфраструктура

Инфраструктура — это скелет, который поддерживает мышцы системы. Мощность инфраструктуры с точки зрения производительности процессора, объема памяти и так далее будет напрямую влиять на способность системы выдерживать высокую нагрузку. Проектирование инфраструктуры для создания высо-

копроизводительной системы само по себе является искусством. Инженеры по инфраструктуре постоянно следят за показателями производительности и применяют их для планирования потребностей приложения в инфраструктуре.

Интеграция с третьими сторонами

При использовании сторонних компонентов и сервисов производительность приложения начинает зависеть и от них. Любая задержка в стороннем компоненте в конечном итоге увеличит задержку самого приложения. Например, как обсуждалось в главе 3, типичное приложение для розничной торговли интегрируется со многими внешними сервисами, такими как системы управления информацией о товарах поставщиков, системы управления складом и т. д., и в таких случаях выбор высокопроизводительных компонентов жизненно важен.

В процессе тестирования производительности учитывайте все эти факторы, чтобы получить достоверные тестовые сценарии. Например, настройте среду тестирования производительности так, чтобы она была максимально похожа на промышленную среду с точки зрения сети, инфраструктуры, географического местоположения и т. д. В противном случае вы не сможете точно оценить производительность!

Ключевые показатели производительности

Измерение или тестирование производительности приложения предполагает сбор ключевых показателей эффективности (Key Performance Indicator, KPI). Постоянное их измерение на протяжении всего цикла разработки поможет команде скорректировать курс раньше и с меньшими усилиями. Перечислю некоторые KPI, которые вы должны отслеживать.

Время отклика

Под временем отклика подразумевается время, необходимое приложению для ответа на запрос пользователя, например точное время для отображения результатов запроса на поиск продукта. Как мы видели ранее, время отклика веб-приложений не должно превышать 3 с, иначе есть риск потерять большую часть клиентов. Обратите внимание на то, что 3 с — это задержка, с которой сталкивается конечный пользователь, следовательно, она включает в себя не только время ответа API, но и время, необходимое для полной загрузки страницы.

Параллелизм и пропускная способность

К веб-сайтам в любой момент может обращаться большое число пользователей со всего мира. Некоторые высокоскоростные приложения, такие как сайты фондовых бирж, обслуживают миллионы транзакций в секунду. Проверка способности приложения обслуживать заданное количество пользователей в данный момент времени называется *оценкой параллелизма*. Например, вы

можете проверить, способно ли приложение ответить в течение 3 с на запросы 500 пользователей, поступившие одновременно.

Термин «одновременно» часто применяется предприятиями и командами разработчиков ПО, но если встать на точку зрения системы, то в действительности она ставит получаемые запросы в очередь и выбирает их один за другим для обработки в параллельных потоках. Следовательно, с этой точки зрения количество одновременных запросов не лучший показатель. Предпочтительнее оценивать *пропускную способность*. Она показывает количество запросов, которые система может обслужить за определенный интервал времени.

Чтобы было понятнее, рассмотрим аналогию с автомобилями, пересекающими очень короткий мост через реку. Допустим, имеется четыре автомобильные полосы. Если предположить, что транспорт движется беспрепятственно, то каждая машина сможет пересечь мост за несколько сотен миллисекунд. Значит, за секунду общее количество машин, проезжающих по мосту, составит от 30 до 40. Значение 30–40 машин в секунду и есть пропускная способность.

Параллелизм и пропускная способность полезны при планировании мощности сервера и часто используются в разных контекстах для принятия эффективных решений.

Доступность

Доступность — это мера способности системы реагировать на нужды конечных пользователей в одних и тех же приемлемых пределах в течение заданного непрерывного периода. Обычно ожидается, что веб-сайты будут доступны круглосуточно и без выходных, за исключением периодов планового обслуживания. Доступность — важный критерий для тестирования, поскольку приложение может работать хорошо в течение первого получаса, а затем снижать производительность из-за утечки памяти, чрезмерного использования мощности инфраструктуры параллельными пакетными заданиями и многих других непредсказуемых причин.

Теперь, когда мы обсудили ключевые показатели эффективности, посмотрим, как их измерять.

Типы тестов производительности

Чтобы измерить ключевые показатели эффективности, необходимы тесты, спроектированные определенным образом. Далее описаны три распространенных типа тестов производительности.

Нагрузочные/объемные тесты

Как обсуждалось ранее, для проверки способности приложения обслуживать ожидаемое количество пользователей в приемлемое время измеряются парал-

лелизм и пропускная способность. Предположим, что вам нужно, чтобы функция поиска возвращала результаты в течение 2 с, обслуживая одновременно 300 пользователей. Тест производительности, имитирующий это количество пользователей и проверяющий, соответствует ли приложение ожидаемому целевому времени отклика, называется *объемным* или *нагрузочным*. Возможно, вам придется повторить такие тесты несколько раз, чтобы обеспечить согласованность и измерить среднее значение.

Стресс-тесты

Обычно производительность приложений снижается с увеличением числа обслуживаемых пользователей. Например, приложение может укладываться в допустимые временные рамки, обслуживая сразу X пользователей, но, когда их число увеличивается, оно начинает отвечать с задержками, и, наконец, когда число пользователей достигает $X + n$, начинают возникать ошибки. Вы должны точно измерить эти цифры. Они помогут при планировании инфраструктуры, масштабировании приложения в новых регионах или во время таких мероприятий, как распродажи. Тест производительности в этом сценарии должен увеличивать нагрузку на приложение постепенно, небольшими шагами, чтобы точно определить момент, когда начнут возникать ошибки. Такое воздействие на систему с целью найти точку разрушения называется *стресс-тестированием*.

Тесты на отказоустойчивость

Когда приложение какое-то время обслуживает ожидаемое количество пользователей, время отклика может увеличиться из-за проблем с инфраструктурой, утечки памяти или других проблем. Тесты, проверяющие способность приложения работать под постоянной нагрузкой в течение длительного времени, называются *тестами на отказоустойчивость*.

При разработке всех этих тестов важно сохранять их реалистичность и избегать перегрузки приложения экстремальными ситуациями, которые могут никогда не произойти. Например, не все пользователи будут входить в приложение в один и тот же момент. Более реалистичный вариант — последовательный вход с перерывами в несколько миллисекунд. Задержка между началом теста и моментом, когда все виртуальные пользователи выполняют вход, называется *темпом*. Тестовые сценарии должны быть близки к реальной практике: например, вы можете запланировать увеличение числа вошедших пользователей до 100 за 1 мин.

Кроме того, пользователи не роботы. Они не могут войти в систему, найти продукт и совершить покупку за миллисекунды, но тестовые сценарии производительности могут быть непреднамеренно разработаны именно таким образом. В действительности пользователям требуется хотя бы несколько секунд между действиями, чтобы поразмыслить, и обычно после входа в систему им нужно несколько минут, чтобы завершить транзакцию, например купить товар. В терминах

тестирования производительности это называется *временем на размышление*. Предусмотрите соответствующее время на размышление в своих тестовых сценариях и распределите действия пользователя на интервал в несколько секунд или минут. Со временем на размышление связана еще одна концепция, называемая *шагом нагрузки*. Она определяет время между транзакциями (не действиями пользователя). В реальной жизни пользователи могут через некоторое время снова инициировать транзакции. Итак, если вы ожидаете, что в часы пик будет совершаться до 1000 транзакций в час, то можете распределить транзакции на интервале 1 ч, настроив времена задержки между транзакциями. Эти три атрибута обязательно нужно настроить с умом, чтобы получить реалистичную оценку производительности приложения.

Типы режимов нагрузки

В предыдущем разделе мы говорили о различных типах тестов производительности, применяемых для измерения ключевых показателей эффективности. На основе этих тестов создаются различные шаблоны нагрузки на приложение с использованием атрибутов, которые мы только что обсудили: времени выхода на рабочий режим, времени на размышление, количества одновременных пользователей и темпа. В этом разделе рассмотрим некоторые шаблоны нагрузки, часто применяемые при тестировании.

Шаблон плавного нарастания нагрузки

В шаблоне постепенного нарастания нагрузки (рис. 8.1) пользователи постепенно наращивают нагрузку в течение заданного периода, а затем достигнутая нагрузка длительное время поддерживается на постоянном уровне для измерения производительности. Это очень распространенная закономерность в реальных сценариях, например при распродажах в «черную пятницу», когда количество пользователей постепенно нарастает и сохраняется на некотором уровне в течение определенного периода, прежде чем пойти на спад.

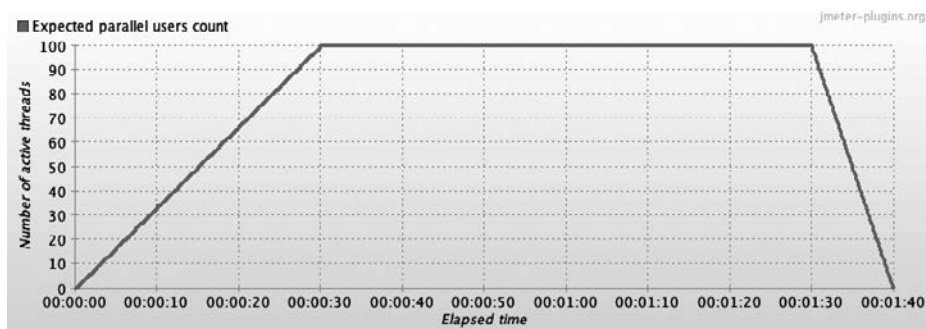


Рис. 8.1. Постепенное увеличение количества пользователей

Шаблон ступенчатого нарастания нагрузки

В шаблоне ступенчатого нарастания нагрузки (рис. 8.2) количество пользователей увеличивается периодически, например, на 100 каждые 2 мин. Измерение производительности приложения для каждого количества шагов увеличения числа пользователей поможет оценить его производительность при различных нагрузках. Шаблон ступенчатого нарастания полезен при настройке производительности и планировании мощности инфраструктуры.



Оценка производительности при различных нагрузках предполагает измерение среднего времени отклика при многократном запуске тестов.

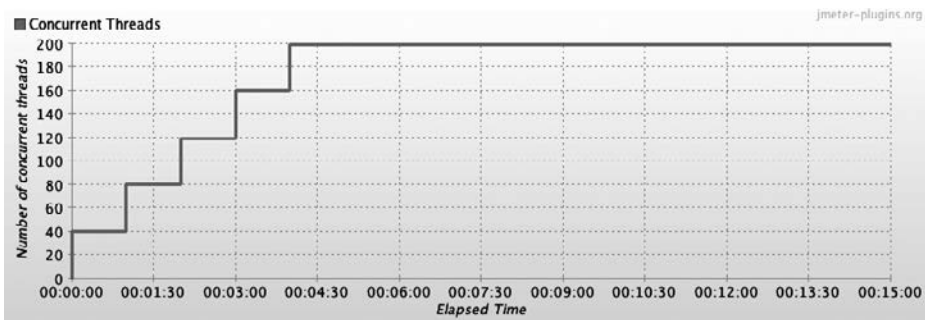


Рис. 8.2. Ступенчатое увеличение количества пользователей

Шаблон «пик — пауза»

Шаблон «пик — пауза» (рис. 8.3) имеет место, когда нагрузка на систему несколько раз подряд увеличивается до пиковой, а затем снижается до полного покоя. Этот сценарий можно наблюдать, например, на сайтах социальных сетей, когда нагрузка циклично возрастает до пиковых значений в течение дня.

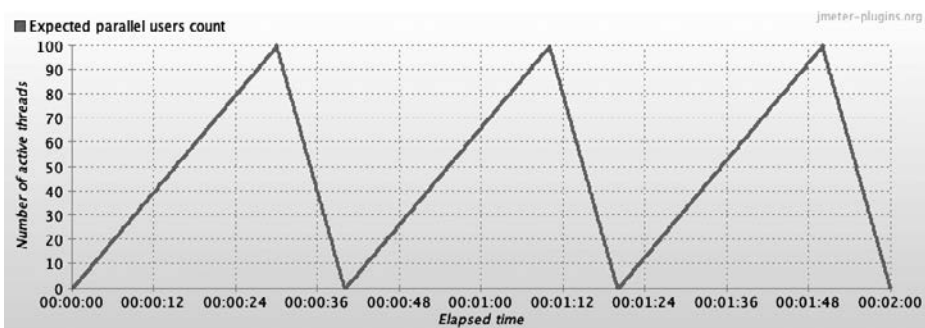


Рис. 8.3. Шаблон изменения нагрузки «пик — пауза»

Инструменты тестирования производительности помогают генерировать эти шаблоны, как мы увидим далее.

Этапы тестирования производительности

Теперь, обсудив ключевые показатели эффективности, типы тестов производительности и шаблоны нагрузки, перейдем к пошаговому описанию этапов тестирования производительности. Это поможет вам спланировать время и мощности, необходимые для тестирования производительности в своем проекте.

Шаг 1. Определение целевых показателей эффективности

Первый шаг — определение целевых показателей эффективности на основе потребностей бизнеса. Лучший способ для этого — рассмотреть их качественно, а затем перевести в цифры¹. Например, размышления о производительности с качественной точки зрения могут привести к таким целям.

- Приложение должно иметь возможность масштабироваться для обслуживания еще одной новой страны.
- Приложение должно работать лучше, чем его конкурент X.
- Новая версия приложения должна работать лучше, чем предыдущая.

Эти качественные цели естественным образом ведут к следующим шагам. Если цель состоит в том, чтобы сделать новую версию лучше предыдущей, то необходимо измерить производительность предыдущей версии и посмотреть, улучшились ли текущие показатели. Аналогично если вы знаете показатели эффективности конкурентов, то вам нужно сравнить с ними свои показатели.



Бизнесмены склонны называть параметры производительности, которые не отражают фактическую модель использования. Всегда извлекайте целевые ключевые показатели эффективности из данных.

- Если есть существующее приложение, проанализируйте его производительность, чтобы получить ключевые показатели эффективности и шаблоны нагрузки.
- Если создаете новое приложение, запросите данные о производительности приложений-конкурентов.
- Если приложение совершенно новое и нет значимых ориентиров, задействуйте данные об использовании Интернета по всей стране, вероятной продолжительности пиковой нагрузки и т. д., чтобы определить целевые ключевые показатели эффективности.

¹ Дополнительную информацию об этой идее см. в статье Скотта Барбера Get Performance Requirements Right — Think Like a User (<https://oreil.ly/D3ujD>).

Шаг 2. Описание тестовых сценариев

Второй шаг — описать тестовые сценарии с использованием шаблонов нагрузки и семантики типов тестов. Тестовые сценарии обязательно должны измерять доступность, пропускную способность и время отклика всех критически важных конечных точек. Эти сценарии впоследствии подскажут, как настроить тестовые данные. Вам может понадобиться всего несколько тестовых сценариев, что выгодно отличает тестирование производительности от функционального тестирования.

Шаг 3. Подготовка среды для тестирования производительности

Как упоминалось ранее, среда тестирования производительности должна быть максимально приближена к промышленной среде. Это необходимо для получения реалистичных результатов и поможет выявить узкие места в конфигурациях среды.

Вот примерный чек-лист параметров для достижения этой цели, который вы можете адаптировать к своим обстоятельствам.

- Соответствующие уровни/компоненты следует развертывать аналогичным образом.
- Конфигурации машин (количество процессоров, объем памяти, версия ОС и т. д.) должны быть одинаковыми.
- Машины должны быть размещены в одном и том же географическом месте в облаке.
- Пропускная способность сети между машинами должна быть одинаковой.
- Конфигурации приложений, такие как ограничение скорости, должны быть одинаковыми.
- Если в фоновом режиме будут выполняться пакетные задания, они должны быть запущены и в тестовой среде. Если приложение отправляет электронные письма, соответствующие системы тоже должны быть установлены в тестовой среде.
- Если в промышленной среде имеются балансировщики нагрузки, они должны быть установлены и в тестовой среде.
- Стороннее программное обеспечение должно быть доступно хотя бы в виде имитаций.

Настройка тестовой среды, сходной по параметрам с промышленной, часто оказывается сложной задачей из-за дополнительных затрат, облачные ресурсы обходятся дешевле. Возможно, вам придется поговорить о стоимости и ценности с заинтересованными сторонами. Если вы не выиграете эту битву, то приготовьтесь пойти на значительный компромисс в определенных частях настройки тестовой среды и дайте понять заинтересованным сторонам, что из-за этих уступок измеренные показатели производительности могут быть ненадежными.



Лучше всего в самом начале проекта запросить настройку тестовой среды вместе со средой контроля качества, чтобы она была доступна, когда понадобится.

Помимо тестовой среды, вам понадобится отдельная машина для запуска тестов производительности. Запланируйте размещение отдельных средств запуска тестов в разном географическом местоположении (это можно организовать с помощью облачных провайдеров), чтобы понаблюдать за поведением с задержками в сети пользователей из нескольких стран, если ваше приложение предназначено для обслуживания глобальной аудитории.

Шаг 4. Подготовка тестовых данных

Тестовые данные, подобно тестовой среде, которая должны быть максимально похожей на промышленную среду, тоже должны максимально точно отражать промышленные данные. Точность результатов измерений показателей производительности будет во многом зависеть от качества тестовых данных, следовательно, это очень важный шаг. Идеально было бы использовать фактические промышленные данные после анонимизации любой конфиденциальной информации, поскольку в этом случае они будут отражать фактический размер и сложность базы данных. Однако в определенных ситуациях это может быть невозможно из соображений безопасности. В таких случаях подготовьте тестовые данные, которые более или менее точно имитируют данные из промышленной среды.

Вот несколько советов по созданию тестовых данных.

- Оцените размер промышленной базы данных (например, 1 Гбайт или 1 Тбайт) и настройте сценарии для заполнения тестовой базы данных фиктивной информацией. Для запуска разных тестов может потребоваться очищать и повторно заполнять тестовую базу данных, поэтому наличие соответствующих сценариев будет иметь решающее значение.
- Создавайте разнообразные тестовые данные, аналогичные тем, что наблюдаются в промышленной среде. Вместо «Рубашка1», «Рубашка2» и так далее используйте реальные значения, аналогичные промышленным, например: «Оливково-зеленая футболка Van Heusen с V-образным вырезом».
- Добавьте изрядную долю ошибочных значений, например адресов с орфографическими ошибками, недостающими значениями и т. д., которые могут имитировать данные, фактически вводимые пользователем.
- Данные должны одинаково распределяться по таким факторам, как возраст, страна и т. д.
- В зависимости от тестовых сценариев вам может потребоваться создать множество уникальных данных, таких как уникальные номера кредитных карт, учетные данные для входа и т. д., чтобы запускать нагрузочные тесты для оценки производительности при обслуживании большого числа пользователей одновременно.

Подготовка тестовых данных может оказаться утомительной работой! Эти действия необходимо планировать заблаговременно. Втиснуть все это в график разработки позднее может быть невозможно, а при попытке сделать так вы можете получить некачественные тестовые данные, что приведет к неточностям в измерении показателей производительности.

Шаг 5. Интеграция инструментов мониторинга

Следующий шаг — интеграция инструментов мониторинга производительности (например, New Relic, Dynatrace, Datadog), чтобы можно было наблюдать за поведением системы во время тестирования производительности. Эти инструменты очень помогают в устранении проблем с производительностью. Например, во время тестирования производительности запросы могут завершиться неудачей из-за нехватки памяти на компьютере, и инструменты АРМ легко выявляют такие проблемы.

Шаг 6. Написание сценария и запуск тестов

Последний шаг — написание сценариев тестирования с помощью инструментов и их запуск в тестовой среде. Существует множество инструментов, которые можно использовать для создания сценариев и запуска тестов одним щелчком кнопкой мыши, а также для их интеграции в CI. В числе наиболее популярных можно назвать JMeter, Gatling, k6 и Apache Benchmark (ab). Помимо этих инструментов с открытым исходным кодом, существуют также коммерческие облачные инструменты, такие как BlazeMeter, NeoLoad и др. Некоторые из них предлагают простые пользовательские интерфейсы для настройки тестов и не требуют программирования. С их помощью можно получать отчеты с результатами и графиками, а коммерческие инструменты даже предлагают информационные панели. В следующем разделе вы найдете упражнение по созданию тестовых сценариев с использованием JMeter и их интеграции с CI.



Для выполнения теста производительности может потребоваться от нескольких минут до нескольких часов в зависимости от его целей и задач. Чтобы получить представление о том, сколько времени займет тестирование, выполните пробный прогон сценариев с небольшим количеством пользователей и только потом запускайте полноценный тест.

Эти шесть шагов тестирования производительности мы выполним в рамках упражнения в следующем разделе. Ключом к успешной реализации всех шагов является адекватное планирование мощности, как упоминалось ранее. При планировании учитывайте время и возможности для сбора отчетов, отладки и устранения проблем с производительностью, а также настройки мощности сервера. Это завершит весь цикл тестирования производительности!

Упражнения

Возьмем в качестве примера приложение для управления онлайн-библиотекой и протестируем его производительность, выполнив последовательно каждый шаг. Для удобства будем считать, что это приложение обладает ограниченным набором простых функций. Оно поддерживает два типа пользователей: администраторов, которые могут добавлять и удалять книги, и клиентов, которые могут просматривать список всех книг и искать книгу по ее идентификатору. Соответствующие REST API: `/addBook`, `/deleteBooks`, `/books` и `/viewBookByID`.

Шаг 1. Определение целевых показателей эффективности

Чтобы определить целевые показатели эффективности для приложения управления библиотекой, предположим, что мы получили от компании и собственной маркетинговой команды следующие данные.

- Они ведут агрессивную кампанию по запуску в двух европейских городах и ожидают, что в первый год у них зарегистрируются 100 000 пользователей.
- Согласно проведенным ими исследованиям, в течение одного сеанса пользователи тратят в среднем 10 мин на поиск книг, просмотр похожих книг и т. д.
- Исследования также показали, что типичный пользователь может брать книгу в среднем два раза в месяц, следовательно, они ожидают, что пользователи будут заходить на сайт два раза в месяц.
- В Европе пользователи активны в Интернете с 10:00 до 22:00 (12 ч) ежедневно.

На основе этих данных можем сделать такие выводы.

- Общее количество пользователей, заходящих на сайт ежемесячно:

$$100\,000 \text{ пользователей} \times 2 \text{ входа в месяц} = 200\,000 \text{ пользователей в месяц.}$$

- Среднее количество пользователей в день:

$$\begin{aligned} 200\,000 \text{ пользователей в месяц} / 30 \text{ дней в месяц} = \\ = 6667 \text{ пользователей в день} \end{aligned}$$

(обратите внимание на то, что в выходные дни пользователей может быть больше, чем в будни, но мы рассчитываем среднее число пользователей в день).

- Среднее количество пользователей в час:

$$\begin{aligned} 6667 \text{ среднее количество пользователей в день} / 12 \text{ часов в день} = \\ = 555 \text{ пользователей в час} \end{aligned}$$

(аналогично активность пользователей в разные часы может быть разной).

- Чтобы учесть пиковые нагрузки, проявим щедрость и округлим предыдущий показатель до 1000 пользователей в час.
- Каждый пользователь задействует веб-сайт в течение сеанса продолжительностью 10 мин, что составляет 0,166 667 ч.
- Количество одновременных пользователей:

$$1000 \text{ пользователей в час пик} \times 0,166 = 166.$$

- Если предположить, что каждый пользователь делает не менее пяти запросов (поиск книг и просмотр списка) за 10-минутный сеанс, то система должна обрабатывать:

$$5 \times 1000 \text{ пользователей в час} = 5000 \text{ запросов в час.}$$

Итак, вот наши целевые показатели эффективности.

- Система должна обслуживать запросы 166 одновременно работающих пользователей не дольше 3 с.
- Пропускная способность системы должна составлять 5000 запросов в час.

Прежде чем продолжить, нам нужно согласовать эти цифры с командой управления клиентами. Также мы могли бы изучить бизнес, оценить его перспективы через год и еще раз уточнить целевые показатели.



Это лишь примерный расчет, который дает представление о том, как находить целевые показатели эффективности. Как упоминалось ранее, в первую очередь желательно исследовать промышленные данные, полученные из существующего приложения, своего или конкурентов, что поможет точнее оценить целевые показатели и шаблон нагрузки.

Шаг 2. Описание тестовых сценариев

Теперь, зная целевые показатели эффективности, можно определить соответствующие тестовые сценарии производительности на основе функций приложения управления библиотекой. Если вспомнить факторы, которые мы обсуждали, то в число тестовых сценариев для нашего приложения могут входить:

- сравнение времени ответа для всех четырех конечных точек — `/addBook`, `/deleteBooks`, `/viewBookById` и `/books`;
- нагрузочное тестирование конечных точек `/viewBookById` и `/books`, ориентированных на одновременное обслуживание 166–200 клиентов, которые должны отвечать на запросы 166 одновременно работающих пользователей не позже чем через 3 с. (Обратите внимание на то, что 3 с включают производительность пользовательского интерфейса, поэтому вам следует выбрать более низкое

граничное значение с учетом особенностей своего приложения.) К двум другим конечным точкам имеют доступ только администраторы, следовательно, им может не потребоваться объемное тестирование;

- стресс-тестирование конечных точек, ориентированных на клиентов, со ступенями увеличения нагрузки по 100 пользователей и определение критических точек;
- проверка пропускной способности 5000 запросов в час. Поток пользователя для этого тестового сценария может включать просмотр списка книг, выбор книги и беглый просмотр ее описания, затем возврат на страницу со списком книг, выбор другой книги и чтение ее описания, снова возврат на страницу со списком книг — всего получается пять запросов на каждый пользовательский поток. Надо не забыть добавить время на размышление, скажем, по 30 с между этими действиями и предположить, что 45 пользователей могут продолжать проходить этот путь в течение часа. Количество вошедших пользователей должно увеличиваться плавно в течение первых 10 мин;
- тест на отказоустойчивость в течение 12 ч, чтобы убедиться, что система постоянно доступна пользователям. Для этого сценария можно повторно применить приведенную ранее схему тестирования пропускной способности и растянуть ее выполнение на 12 ч.

Шаги 3–5. Подготовка данных, среды и инструментов

Для этого упражнения я разработала пример приложения управления библиотекой и разместила его в Heroku. Чтобы выполнить его самостоятельно, можете создать заглушку (как это сделать, рассказывается в пункте «WireMock» в главе 2) для конечной точки `/books` на своем локальном компьютере, как показано в примере 8.1, и настроить ее так, чтобы она возвращала 50 книг. Проверьте заглушку один раз после настройки.



Нагрузочное тестирование общедоступных API может рассматриваться как DDoS-атака, поэтому и необходимо создать заглушку для этого упражнения. В то же время различные инструменты тестирования производительности, такие как JMeter и Gatling, предоставляют тестовые площадки, которые можно использовать для оттачивания навыков тестирования производительности. Ссылки на такие площадки вы найдете на официальных сайтах инструментов. Проводите тестирование с минимальной предписанной нагрузкой.

Пример 8.1. Конечная точка `/books`

GET: `/books`

Response:

Status Code: 200

```
Body:
[
{ "id": 1,
  "name": "Man's search for meaning",
  "author": "Victor Frankl",
  "Language": "English",
  "isbn": "ABCD1234"
},
{ "id": 2,
  "name": "Thinking Fast and Slow",
  "author": "Daniel Kahneman",
  "Language": "English",
  "isbn": "UFGH1234"
}]
```

Шаг 6. Написание тестовых сценариев и их запуск с помощью JMeter

JMeter — популярный инструмент тестирования производительности с открытым исходным кодом, поддерживающий возможность интеграции с CI и создающий красивые графические отчеты. Он также интегрируется с BlazeMeter — облачным инструментом анализа производительности, что может пригодиться тем, кто желает избавиться от хлопот, связанных с управлением инфраструктурой. JMeter написан на Java и имеет сообщество активных разработчиков, которые вносят свой вклад в создание различных ценных плагинов. Рисунки в разделе «Типы режимов нагрузки» ранее в этой главе созданы с использованием одного из них. Существуют также хорошая документация (<https://oreil.ly/Kt2YT>) и обучающие руководства для начинающих. Давайте установим этот инструмент и напишем несколько тестовых сценариев для нашего приложения.

Установка и настройка

Выполните следующие шаги, чтобы установить и настроить JMeter.

1. Загрузите ZIP-файл с официального сайта (<https://oreil.ly/0kKfX>) и установите его. Убедитесь в совместимости с вашей локальной версией Java. Также убедитесь, что в настройках профиля в `bash_profile` устанавливается переменная `JAVA_HOME`.
2. Чтобы открыть графический интерфейс JMeter, запустите в терминале сценарий оболочки `jmeter.sh`, находящийся в папке `/apacheJMeter-version/bin`.
3. Мы также будем использовать плагины JMeter. Вы можете загрузить менеджер плагинов Plugins Manager с официального сайта (<https://oreil.ly/fIhE0>) и поместить JAR-файл в папку `/apacheJMeter-version/lib/ext`.
4. Перезапустите JMeter. После этого в меню Options (Параметры) должен появиться пункт Plugins Manager (Диспетчер плагинов).

Рабочий процесс

Выполните шаги, описанные далее, чтобы создать и настроить простой тест JMeter для оценки времени ответа конечной точки `/books`.

1. Создайте группу потоков выполнения в графическом интерфейсе JMeter, щелкнув правой кнопкой мыши на пункте **Test Plan** (План тестирования) на панели слева и выбрав **Add ▸ Threads (Users) ▸ Thread Group** (Добавить ▸ Потоки (Пользователи) ▸ Группа потоков). Дайте группе потоков имя **ViewBooks**. Настройте параметры так, как показано на рис. 8.4 (**Number of Threads** (Количество потоков) — 1, **Ramp-up period** (Темп) — 0, **Loop Count** (Число циклов) — 10), чтобы получить десять измерений времени ответа конечной точки и усреднить их.

Thread Group

Name:

Comments:

Action to be taken after a Sampler error

☒ Continue ☐ Start Next Thread Loop ☐ Stop Thread ☐ Stop Test ☐ Stop Test Now

Thread Properties

Number of Threads (users):

Ramp-up period (seconds):

Loop Count: ☐ Infinite

Рис. 8.4. Настройка группы потоков для выполнения одного запроса 10 раз

2. Добавьте сэмплер, выполняющий HTTP-запросы к API. Щелкните правой кнопкой мыши на только что созданной группе потоков на панели слева и выберите **Add ▸ Sampler ▸ HTTP Request** (Добавить ▸ Сэмплер ▸ HTTP-запрос). Укажите имя веб-сервера, тип HTTP-запроса и путь (рис. 8.5). Дайте сэмплеру имя `viewBooksRequest`.
3. Добавьте прослушиватели, которые будут фиксировать каждый запрос и ответ во время выполнения теста. Щелкните правой кнопкой мыши на сэмплере `viewBooksRequest` и выберите **Add ▸ Listeners ▸ View Results Tree** (Добавить ▸ Прослушиватели ▸ Просмотр дерева результатов), затем повторите процесс, но на этот раз выберите прослушиватель **Aggregate Report** (Совокупный отчет).
4. Сохраните основу теста. Затем, чтобы измерить время ответа, щелкните на кнопке **Run** (Выполнить). Результаты будут доступны в разделах слушателей.

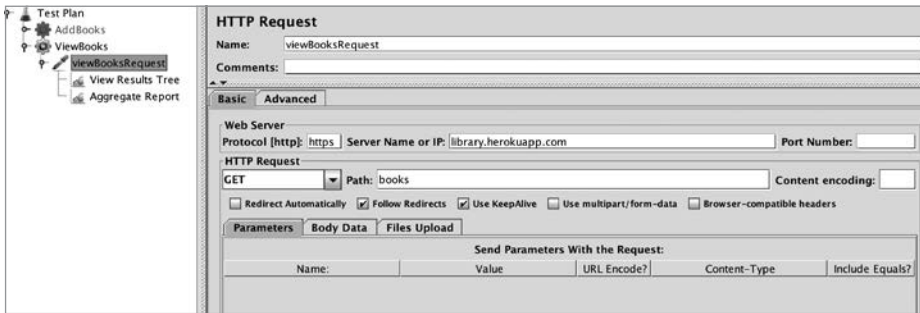


Рис. 8.5. Настройки сэмплера viewBooksRequest, выполняющего HTTP-запросы

Выберите пункт View Results Tree (Просмотр дерева результатов) на панели слева, чтобы просмотреть выходные данные этого прослушивателя. Вы увидите список запросов, отправленных JMeter, с признаком успеха или неудачи для каждого. JMeter воспринимает код состояния ответа 200 как успех, а все остальные коды — как неудачу. Следует отметить, что в вашем приложении могут возникнуть ситуации, когда сервис вернет код состояния 200, сообщаящий об успехе операции, которая не дала ожидаемых результатов. Например, конечная точка /addBook может вернуть код состояния 200 в ответ на попытки добавить уже имеющиеся книги с сообщением о том, что это дубликат. В таких случаях необходимо добавить явные проверки утверждений (проверки утверждений, как и прослушиватели, являются компонентами JMeter). Щелчком на любом запросе в представлении View Results Tree (Просмотр дерева результатов) можно вывести данные из запроса и ответа, которые могут пригодиться для дальнейшей отладки (рис. 8.6).

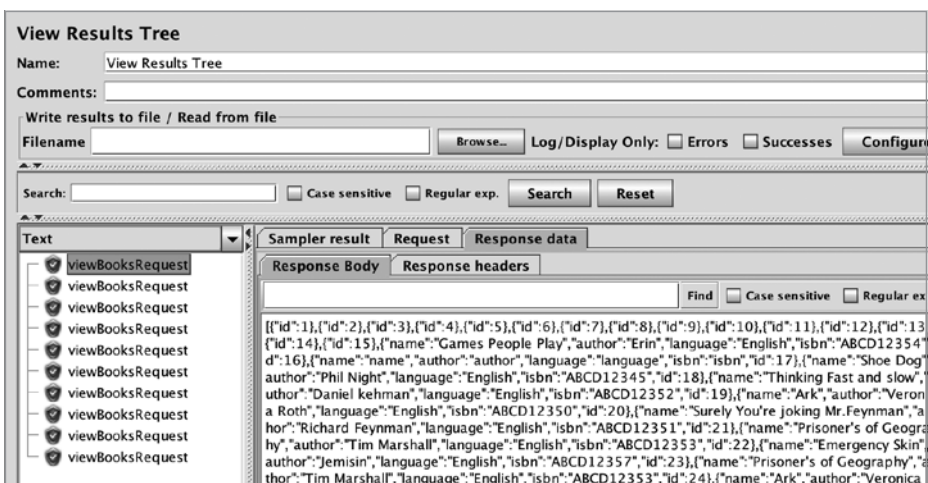


Рис. 8.6. Вывод прослушивателя View Results Tree (Просмотр дерева результатов)

Аналогично если выбрать **Aggregate Report** (Совокупный отчет), то вы увидите таблицу с такими показателями, как среднее значение, медиана, пропускная способность и т. д. Для конечной точки `/books` среднее время ответа по десяти циклам тестирования составляет 379 мс (рис. 8.7) — это лучшее время отклика, когда приложение не находится под нагрузкой.

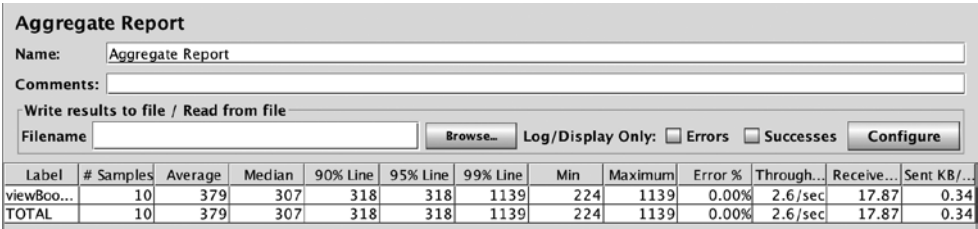


Рис. 8.7. Представление **Aggregate Report** (Совокупный отчет) с результатами тестирования времени ответа конечной точки `/books`

Следующий шаг — нагрузочное тестирование конечной точки `/books` со 166 одновременно работающих пользователями и проверка времени ответа. JMeter предлагает множество способов настройки шаблонов нагрузки. Здесь рассмотрим три наиболее простых варианта.

Как мы уже видели, в JMeter есть базовый элемент — группа потоков, в котором можно размещать прослушиватели и контроллеры. Его можно использовать также для настройки параметров нагрузочного тестирования, таких как количество параллельных потоков, темп и количество повторений теста. Ранее мы настроили группу потоков **ViewBooks** для выполнения запроса к `/books` десять раз, чтобы оценить время ответа. Теперь, чтобы провести нагрузочное тестирование, изменим параметры так: **Number of Threads** (Количество потоков) — 166, **Ramp-up period** (Темп) — 0, **Loop Count** (Количество циклов) — 5. JMeter запустит 166 потоков выполнения одновременно и выполнит цикл тестирования пять раз, чтобы получить пять замеров и усреднить их.

Также можно воспользоваться удобным плагином, открывающим доступ к дополнительным типам групп потоков, с помощью которых можно настроить различные шаблоны нагрузки, такие как шаблон ступенчатого нарастания. Далее я покажу, как использовать группы **Concurrency Thread Group** (Группа параллельных потоков) и **Ultimate Thread Group** (Универсальная группа потоков). Начнем с группы параллельных потоков, которая предоставляет контроллер параллелизма для нагрузочного тестирования.

1. Выберите в меню пункт **Options ▶ Plugins Manager** (Параметры ▶ Диспетчер плагинов). На вкладке **Available Plugins** (Доступные плагины) найдите **Custom Thread Groups** (Пользовательские группы потоков) и установите его.

2. Перезапустите JMeter, чтобы новые типы групп потоков стали доступными.
3. Щелкните правой кнопкой мыши на Test Plan (План тестирования) на панели слева и выберите Add ▸ Threads (Users) ▸ bzm ▸ Concurrency Thread Group (Добавить ▸ Потоки (пользователи) ▸ bzm ▸ Группа параллельных потоков).
4. Настройте параметры нагрузки так, как показано на рис. 8.8 (Target Concurrency (Целевой параллелизм) — 166, Ramp Up Time (Время выхода на рабочий режим) — 0.5, Hold Target Rate Time (Темп) — 2). Согласно этим настройкам JMeter увеличит количество пользователей до 166 за 30 с и будет удерживать каждого из них в системе в течение 2 мин.
5. Добавьте в эту группу потоков сэмплер с HTTP-запросом, как раньше, запустите тест и откройте результаты в прослушивателях.

bzm - Concurrency Thread Group

Name:

Comments:

Action to be taken after a Sampler error

☒ Continue
 ☐ Start Next Thread Loop
 ☐ Stop Thread
 ☐ Stop Test
 ☐ Stop Test Now

Target Concurrency:

Ramp Up Time (min):

Ramp-Up Steps Count:

Hold Target Rate Time (min):

Рис. 8.8. Применение Concurrency Thread Group (Группа параллельных потоков) для нагрузочного тестирования конечной точки /books

Помимо других типов, плагин Custom Thread Groups (Пользовательские группы потоков) предоставляет также тип Ultimate Thread Group (Универсальная группа потоков), обладающий дополнительными возможностями. Например, он позволяет настроить начальную задержку перед запуском теста, время выключения после запуска теста и многое другое. Чтобы использовать Ultimate Thread Group (Универсальная группа потоков) для нагрузочного тестирования, выполните следующие шаги.

1. Щелкните правой кнопкой мыши на Test Plan (План тестирования) и выберите Add ▸ Threads (Users) ▸ jp@gc Ultimate Thread Group (Добавить ▸ Потоки (пользователи) ▸ jp@gc Универсальная группа потоков).
2. Настройте параметры тестирования так, как показано на рис. 8.9 (Start Threads Count (Количество запускаемых потоков) — 166, Initial Delay (Начальная задержка) — 0, Startup Time (Время запуска) — 10, Hold Load For (Удерживать нагрузку) — 60, Shutdown Time (Продолжительность остановки) — 10). С этими

настройками JMeter запустит 166 параллельных потоков в течение 10 с и будет удерживать такую нагрузку в течение 1 мин, после чего уменьшит количество пользователей до нуля в течение 10 с. Также можно добавить дополнительные строки, если это необходимо, чтобы создать шаблон «пик — пауза».

- 3. Добавьте сэмплер с HTTP-запросом, как раньше, запустите тест и откройте результаты.

jp@gc – Ultimate Thread Group

Name:

Volume Test ViewBooks

Comments:

Action to be taken after a Sampler error

☒ Continue

☐ Start Next Thread Loop

☐ Stop Thread

☐ Stop Test

☐ Stop Test Now

Threads Schedule

Start Threads Count	Initial Delay, sec	Startup Time, sec	Hold Load For, sec	Shutdown Time
166	0	10	60	10

Add Row

Copy Row

Delete Row

Рис. 8.9. Использование Ultimate Thread Group (Универсальная группа потоков) для нагрузочного тестирования конечной точки /books

На рис. 8.10 показаны результаты, полученные с помощью простой группы параллельных потоков (первый вариант) со 166 пользователями и нулевым временем выхода на рабочий режим, усредненные по пяти попыткам: Average (Среднее) — 801 мс, 90 % Line (90 % уровень) — 1499 мс. Другими словами, 90 % из 166 одновременно работающих пользователей получают ответ примерно через 1,5 с, а в среднем все 166 одновременно работающих пользователей получают ответ в течение 0,8 с. Среднее значение ниже, поскольку, как видно из таблицы, минимальное время получения ответа у некоторых пользователей составило всего 216 мс.

Aggregate Report

Name:

Aggregate Report

Comments:

Write results to file / Read from file

Filename

Browse...

Log/Display Only:

☐ Errors

☐ Successes

Configure

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Through...	Receive...	Sent KB/...
view800...	830	801	731	1499	1543	1611	216	2112	0.00%	177.8/s...	2925.18	23.27
TOTAL	830	801	731	1499	1543	1611	216	2112	0.00%	177.8/s...	2925.18	23.27

Рис. 8.10. Результаты нагрузочного тестирования конечной точки /books

Разработка других сценариев тестирования производительности

В предыдущем разделе мы рассмотрели некоторые способы распределения нагрузки с помощью JMeter. Этот опыт использования инструмента для нагрузочного тестирования станет для вас отличным началом и позволит моделировать другие сценарии тестирования производительности, такие как стресс-тесты, тесты на отказоустойчивость, проверка пропускной способности и т. д. Для проведения стресс-тестирования можно задействовать группу параллельных потоков, чтобы поэтапно увеличивать нагрузку по x пользователей до максимального предела, выполняя каждый шаг в течение заданного времени. Цель в данном случае — найти нагрузку, при которой время отклика замедляется и в конечном итоге приводит к ошибкам.

Чтобы провести тестирование на отказоустойчивость, можете симитировать постоянную нагрузку в течение длительного времени, задействуя универсальную группу потоков. Для проверки почасовой пропускной способности можно воспользоваться плагином **Parallel Controller** (Параллельный контроллер) (<https://oreil.ly/UtuXj>), который параллельно выполняет несколько HTTP-запросов с паузами, определяемыми компонентами **Timer** (Таймер), чтобы, например, симитировать время на размышления. Существует также **Constant Throughput Timer** (Таймер постоянной пропускной способности), который можно применять для фиксации постоянного значения пропускной способности и проверки правильной работы приложения. Он автоматически замедляет количество запросов, посылаемых инструментом JMeter к серверу, если оно превышает установленное значение.

В JMeter есть много других компонентов, помогающих моделировать сценарии использования конкретных приложений. Контроллеры **If**, **Loop** и **Random** позволяют добавлять условия в тесты. Также предусмотрена возможность ввода данных из внешнего источника данных, например из файла CSV. Она называется *тестированием производительности, управляемым данными*. Эту функцию JMeter можно использовать и для настройки тестовых данных в начале теста. Пример такого подхода мы рассмотрим далее.

Тестирование производительности, управляемое данными

Допустим, конечная точка `/addBook` в приложении управления библиотекой принимает запрос с названием книги, автором, языком и ISBN. Чтобы создать нагрузку на эту конечную точку, нужно добавлять уникальные данные в каждый запрос. Для этого можно использовать возможности JMeter по тестированию производительности, управляемом данными, как описано далее.

1. Создайте файл CSV с полями `name`, `author`, `language` и `isbn`. JMeter будет обращаться к ним при определении входных переменных. Добавьте 50 записей с информацией о книгах. (Это можно сделать в Google Sheets и выгрузить файл в формате CSV.)
2. В JMeter добавьте группу потоков и сэмплер с HTTP-запросом к конечной точке `/addBook`. Установите **Loop Count** (Число циклов) равным 50.

3. Чтобы связать CSV-файл и сэмплер с HTTP-запросом, щелкните правой кнопкой мыши на Thread Group (Группа потоков) и выберите Add ► Config Element ► CSV Data Set Config (Добавить ► Элемент конфигурации ► Конфигурация набора данных CSV). В окне CSV Data Set Config (Конфигурация набора данных CSV) укажите путь к файлу CSV и переменные для чтения из файла (рис. 8.11).

CSV Data Set Config

Name: CSV Data Set Config

Comments:

Configure the CSV Data Source

Filename: /pathToInputFile/BooksTestData - Sheet1.csv

File encoding:

Variable Names (comma-delimited): name,author,language,isbn

Ignore first line (only used if Variable Names is not empty): False

Delimiter (use '\t' for tab):

Allow quoted data?: False

Recycle on EOF?: True

Stop thread on EOF?: False

Sharing mode: All threads

Рис. 8.11. Конфигурация набора данных CSV для тестирования, управляемого данными

4. В теле HTTP-запроса к конечной точке /addBook используйте переменные в формате `${имя_переменной}` (рис. 8.12). С помощью той же нотации `${имя_переменной}` к ним можно обращаться везде в тестах JMeter, где это может понадобиться.

HTTP Request

Name: AddBooks

Comments:

Basic Advanced

Web Server

Protocol [http]: https Server Name or IP: library.herokuapp.com

HTTP Request

POST Path: books

☐ Redirect Automatically ☒ Follow Redirects ☒ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compat

Parameters Body Data Files Upload

1 {"name": "\${name}", "author": "\${author}", "language": "\${language}", "isbn": "\${isbn}"}

Рис. 8.12. Ссылки на переменные в файле CSV

Этот тест можно запустить для создания тестовых данных перед запуском тестов производительности.

Интеграция в CI

Последний шаг — интеграция тестов JMeter в конвейер CI в виде отдельной задачи и реализация раннего тестирования производительности. Чтобы получить правильные показатели, важно убедиться, что тесты производительности выполняются в полной изоляции. Для интеграции тестов в CI сохраните их, найдите сохраненные файлы `.jmx` и выполните следующую команду:

```
$ jmeter -n -t <library.jmx> -l <лог-файл> -e -o <Путь к папке  
для сохранения результатов>
```

Используя дополнительные расширения, можно настроить JMeter для предоставления исчерпывающих отчетов на информационной панели (<https://oreil.ly/yBzw0>).

Как видите, JMeter упрощает тестирование производительности, предлагая простой графический интерфейс для настройки и запуска тестовых сценариев.

Дополнительные инструменты тестирования

Существует еще несколько инструментов, которые могут помочь в написании сценариев тестирования производительности. По сути, они предоставляют различные возможности для настройки четырех ключевых параметров шаблонов нагрузки (темп, время на размышление, количество одновременно работающих пользователей и шаг нагрузки). Например, как мы видели ранее, JMeter предлагает графический интерфейс, Gatling предоставляет предметно-ориентированный язык, а Apache Benchmark (ab) использует простые аргументы командной строки. Давайте вкратце познакомимся с Gatling и ab.

Gatling

Gatling (<https://gatling.io/docs>) предоставляет предметно-ориентированный язык (Domain-Specific Language, DSL) на основе Scala для настройки шаблона загрузки. Этот инструмент с открытым исходным кодом позволяет описывать пользовательские сценарии. Тесты можно интегрировать в конвейеры CI. Для знающих язык Scala этот инструмент станет надежным средством моделирования сложных шаблонов нагрузки. В примере 8.2 показан пример сценария на Scala, демонстрирующий, как организовать нагрузку, включающую время на размышления, для конечной точки `/books` приложения управления библиотекой.

Пример 8.2. Пример сценария Scala для нагрузочного тестирования

```
package perfTest

import scala.concurrent.duration._
import io.gatling.core.Predef._
import io.gatling.http.Predef._
```

```

class BasicSimulation extends Simulation {

// Определение HTTP-запроса
val httpProtocol = http
    .baseUrl("https://library.herokuapp.com/")
    .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8")
    .doNotTrackHeader("1")
    .acceptLanguageHeader("en-US,en;q=0.5")
    .acceptEncodingHeader("gzip, deflate")
    .userAgentHeader("Mozilla/5.0 (Windows NT 5.1; rv:31.0) Gecko/20100101
        Firefox/31.0")

// Определение пользовательского сценария, включающего время на размышление
val scn = scenario("BasicSimulation")
    .exec(http("request_1")
        .get("/books"))
    .pause(5) // Think time

// Настройка нагрузки: 166 пользователей,
// одновременно выполняющих сценарий, определенный ранее
setUp(
    scn.inject(atOnceUsers(166))
).protocols(httpProtocol)
}

```

Apache Benchmark

Если вы просто хотите быстро получить некоторые данные о производительности своего приложения, то `ab` (<https://oreil.ly/tDoiU>) станет отличным выбором. Это простой инструмент командной строки с открытым исходным кодом. Если вы используете Mac, то `ab` уже входит в состав операционной системы, поэтому не придется беспокоиться о его установке. Чтобы провести нагрузочное тестирование конечной точки `/books`, имитируя одновременную работу 200 пользователей, запустите из терминала следующую команду:

```
$ ab -n 200 -c 200 https://library.herokuapp.com/books
```

В ответ получите следующие результаты:

```

Concurrency Level:      200
Time taken for tests:    5.218 seconds
Complete requests:      200
Failed requests:         0
Total transferred:      1389400 bytes
HTML transferred:       1340800 bytes
Requests per second:    38.33 [#/sec] (mean)
Time per request:       5217.609 [ms] (mean)
Time per request:       26.088 [ms] (mean, across all concurrent requests)
Transfer rate:          260.05 [Kbytes/sec] received

```


Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	869	2074 97.6	2064	2289
Processing:	249	1324 299.4	1303	1783
Waiting:	249	1324 299.5	1303	1781
Total:	1192	3398 354.3	3370	4027

Percentage of the requests served within a certain time (ms)

50%	3370
66%	3483
75%	3711
80%	3776
90%	3863
95%	3889
98%	4016
99%	4022
100%	4027 (longest request)

Теперь у вас есть представление о возможностях написания сценариев тестирования с использованием разных инструментов и измерении ключевых показателей эффективности бэкенда приложения. Однако на этом тестирование производительности не заканчивается. Если вы обнаружите какие-либо проблемы с производительностью, то придется отыскать и устранить их причины и протестировать снова!

Здесь мы довольно подробно рассмотрели тестирование производительности бэкенда, но еще не закончили. Далее сосредоточимся на тестировании производительности фронтенда!

Введение в тестирование производительности фронтенда

Инструменты тестирования производительности позволяют имитировать поведение приложений в часы пик, но между измеренными и фактическими показателями производительности, наблюдаемыми пользователем, есть разница. Причина в том, что инструменты, описанные ранее, не являются настоящими браузерами и не решают всех задач, которые решает обычный браузер!

Чтобы заполнить этот пробел, исследуем поведение браузера. Как мы видели в главе 6, код внешнего интерфейса, который отображается в браузере, состоит из трех частей, таких как:

- HTML-код, определяющий базовую структуру веб-сайта;
- CSS-код, определяющий стили оформления страницы;
- сценарии, определяющие динамическую логику поведения страницы.

Типичный браузер загружает сначала весь HTML-код с сервера, затем таблицу стилей, изображения и так далее и начинает выполнять сценарии в порядке их упоминания в HTML. При этом некоторые операции, такие как загрузка изображений с разных хостов, могут выполняться параллельно. Но сценарии всегда выполняются последовательно, потому что сценарий может полностью изменить способ отображения страницы. Поскольку сценарии могут находиться в конце HTML, страница становится видимой пользователю, только когда весь документ полностью загружен и обработан.

Инструменты тестирования, рассмотренные ранее, не решают этих задач. Они напрямую обращаются к серверу и получают HTML-код, но не отображают страницу. Поэтому, даже если время ответа сервисов составляет меньше 1 с, конечный пользователь может увидеть страницу со значительной задержкой из-за дополнительных задач отображения, которые должен решить браузер. По оценкам, на такие задачи отображения приходится 80–90 % (<https://oreil.ly/spWi1>) всего времени загрузки страницы — удивительно, не правда ли?

Например, если перейти на домашнюю страницу CNN (<http://www.cnn.com>), то браузер выполнит 90 задач, прежде чем страница появится перед вами. На рис. 8.13 показаны первые 33 из этих задач. Если вы думали, что оптимизация времени ответа веб-сервиса сама по себе существенно повлияет на производительность веб-сайта, то вот вам доказательство, которое может изменить это мнение!

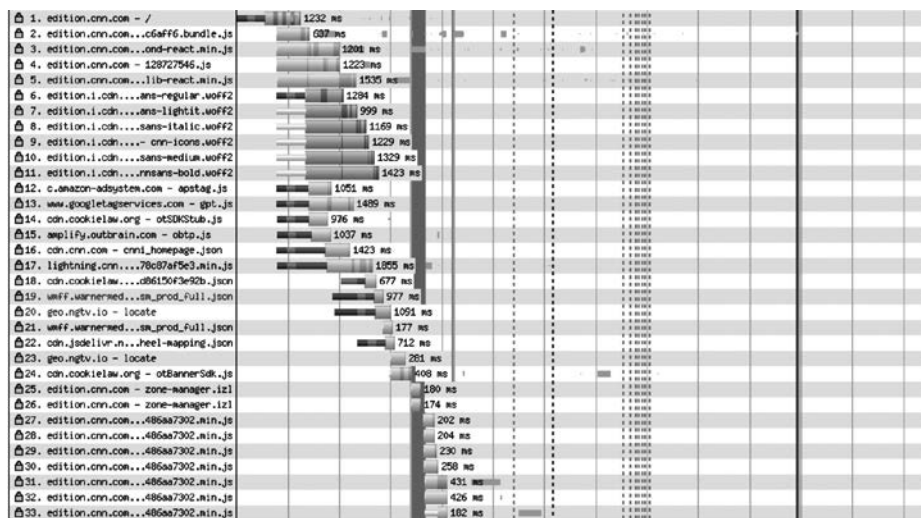


Рис. 8.13. Задачи, выполняемые браузером при отображении домашней страницы CNN

Однако ключевые показатели эффективности, описанные и измеренные в рамках выполненных ранее упражнений, по-прежнему актуальны и важны. Они чрезвычайно важны для планирования емкости системы и устранения проблем с про-

изводительностью. Другими словами, они помогают ответить на такие вопросы: «Выдержит ли приложение пиковую нагрузку в 5000 транзакций во время распродаж в “черную пятницу”?» Но даже если ключевые показатели эффективности сообщают, что максимальное время отклика приложения составляет ~1,5 с, это может не отражать того, что реально наблюдает конечный пользователь. Поэтому вы должны оценить также показатели производительности пользовательского интерфейса. Именно об этом и поговорим далее.

Для начала разберемся с основными факторами, влияющими на производительность пользовательского интерфейса, и с показателями, которые необходимо измерить для ее количественной оценки.

Факторы, влияющие на производительность интерфейса

На производительность внешнего интерфейса влияют несколько факторов.

Сложность кода пользовательского интерфейса

Отказ от применения передовых практик, таких как уменьшение размера исходного кода на JavaScript, уменьшение количества HTTP-запросов, выполняемых страницей, и внедрение методов кэширования, влечет за собой ухудшение производительности. Например, серверу требуется как минимум несколько миллисекунд, чтобы ответить на каждый HTTP-запрос, и если странице приходится делать много таких запросов, то задержки будут накапливаться.

Сети доставки контента (Content Delivery Network, CDN)

CDN — это совокупность серверов, размещенных в разных местах по всему миру, чтобы обеспечить более эффективную доставку веб-контента, например изображений, пользователям. Как обсуждалось ранее, географическое местоположение сервера и пользователя влияет на производительность приложения из-за задержек в сети. Чтобы уменьшить задержку в сети, контент хранится в CDN и обслуживается с сервера, который физически находится ближе к пользователю. Это гораздо проще, чем разворачивать копии приложения в разных географических точках, однако производительность самой CDN тоже влияет на время загрузки страницы.

DNS-запросы

Обычно браузеру требуется 20–120 мс для поиска IP-адреса по имени хоста. Это известно как разрешение в системе доменных имен (Domain Name Service, DNS). Сделав это в первый раз, браузер и ОС кэшируют IP-адрес, сокращая время загрузки страницы при последующих посещениях. Интернет-провайдер (Internet Service Provider, ISP) тоже на некоторое время кэширует IP-адреса, что способствует повышению производительности. Однако на первое обращение пользователя к приложению влияет время поиска DNS.

Задержка сети

Пропускная способность сети пользователя оказывает огромное влияние на общее время загрузки страницы. Как вы видели в главе 6, данные, собранные по всему миру, показывают, что в настоящее время мобильные устройства применяются гораздо чаще, чем настольные компьютеры, а пропускная способность мобильной сети временами может быть очень низкой как в городских, так и в сельских районах. Некоторые сайты стараются решить эту проблему, предлагая облегченную версию, когда определяют, что пропускная способность низкая. Однако пользователи, действующие в сетях с малой пропускной способностью (например, 3G), привыкли к их медлительности и не жалуются, если только производительность не оказывается хуже некуда.

Кэширование в браузере

Помимо IP-адресов, браузер кэширует много другого контента (изображения, cookie-файлы и т. д.) после первого посещения, поэтому время загрузки страницы при первом и последующих обращениях к ней часто значительно различается. Кэширование в браузере можно включить намеренно, чтобы сократить время загрузки страницы.

Передача данных

Если между пользователем и приложением передаются большие объемы данных, то очевидно, что это повлияет на общую производительность внешнего интерфейса из-за задержки в сети.

Глядя на все эти факторы, вы можете подумать, что практически ни один из них нельзя оптимизировать, так с чего же тогда начать? Многие в индустрии разработки ПО тоже чувствовали свое бессилие, столкнувшись с этой проблемой. Но им в помощь была придумана модель RAIL.

Модель RAIL

Модель RAIL (<https://oreil.ly/cKuz1>) помогает организовать мыслительный процесс вокруг производительности пользовательского интерфейса. Она разработана с учетом руководящего принципа, согласно которому в основе производительности пользовательского интерфейса лежит взаимодействие с конечным пользователем и количественные показатели производительности должны определяться именно для этих взаимодействий. Часто полезно посмотреть на производительность пользовательского интерфейса через эту призму и интегрировать целевые показатели в усилия по тестированию.

Модель RAIL делит взаимодействие пользователя с веб-сайтом на четыре ключевые области.

Ответ (response)

Приходилось ли вам оказываться в ситуации, когда вы нажимали кнопку, но не видели никаких внешних проявлений, подтверждающих это, что заставляло вас задуматься: а действительно ли вы нажали ее? Эта задержка известна как *задержка ввода*. Ответ в RAIL определяет целевые значения для задержки ввода. Когда пользователь выполняет действие на веб-сайте, например нажимает кнопку, переключает элемент, устанавливает флажок и т. д., RAIL требует, чтобы время ответа на такое действие было менее 100 мс, в противном случае пользователь почувствует задержку!

Анимация (animation)

Аналогично пользователь будет воспринимать задержку в анимационных эффектах (например, индикаторах загрузки, прокрутке, перетаскивании и т. д.), если каждый кадр не будет успевать завершать выполнение за 16 мс (минимум, необходимый для достижения частоты 60 кадров в секунду).

Простой (idle)

Общий шаблон проектирования пользовательского интерфейса, суть которого заключается в группировке некритических задач, таких как возврат аналитических данных, загрузка поля комментариев и т. д., и выполнении их позже, во время простоя браузера. В идеале эти задачи должны объединяться в блоки, на выполнение которых уходит около 50 мс, чтобы, когда пользователь вернется к взаимодействию, вы могли ответить в течение 100 мс.

Нагрузка (load)

Высокопроизводительный сайт должен стремиться начать отображать страницу в течение 1 с, потому что только тогда пользователи почувствуют, что полностью контролируют навигацию (согласно исследованию, упомянутому ранее).

Как видите, модель RAIL заставляет задуматься о том, что тестировать в пользовательском интерфейсе. Она также обеспечивает конкретный язык для общения внутри команд вместо выражения смутных ощущений типа «страница кажется медленной»!

Показатели производительности пользовательского интерфейса

На практике высокоуровневые цели, установленные моделью RAIL, разбиваются на более мелкие показатели, чтобы точнее настроить отладку проблем с производительностью. В отрасли принят следующий набор стандартных показателей производительности пользовательского интерфейса.

Отрисовка первого содержимого

Время, необходимое браузеру для отрисовки первого элемента из DOM (изображений, небелых элементов, графики SVG и т. д.). Это время помогает понять, как долго пользователь должен ждать, чтобы увидеть хоть что-нибудь после ввода адреса веб-сайта.

Время до интерактивности

Время до начала поддержки интерактивных действий на странице. В стремлении повысить визуальную привлекательность страницы многие старались сделать ее элементы видимыми как можно быстрее, но те могли не реагировать на действия пользователя, что вызывало недоумение. Следовательно, вместе с измерением времени до появления первого контента на странице этот показатель помогает понять, полезна ли представленная информация или это просто шум.

Отрисовка наиболее заметного элемента содержимого

Время, необходимое для отрисовки самого заметного элемента на веб-странице, например большого фрагмента текста или изображения.

Суммарное смещение макета

Приходилось ли вам видеть сайты, на которых страница смещалась вниз по мере загрузки дополнительного контента уже после того, как вы начали читать статью, из-за чего вы теряли место, где читали? Это здорово раздражает, не так ли? Цель этого показателя — измерение визуальной стабильности страницы и количественная оценка того, как часто пользователю приходится наблюдать неожиданные изменения ее макета. Чем меньше это число, тем лучше производительность.

Задержка первого взаимодействия

Между отрисовкой первого содержимого и временем начала взаимодействия, когда пользователь может щелкнуть на ссылке или выполнить какое-то другое действие на странице, есть промежуток времени, в течение которого страница продолжает загружаться. Этот показатель отражает временную задержку до момента, когда можно осуществить первое взаимодействие.

Максимальная потенциальная задержка первого взаимодействия

Это наихудший сценарий задержки первого взаимодействия. Она соответствует времени, затраченному на самую длительную задачу, которая выполняется между отрисовкой первого содержимого и моментом, когда становится возможно первое взаимодействие.

Google классифицирует показатели отрисовки наиболее заметного элемента содержимого, задержки первого взаимодействия и совокупного изменения макета как наиболее важные (<https://web.dev/vitals>), стремясь помочь нетехническим специалистам понять, из чего складывается производительность сайта. Боль-

шинство инструментов тестирования производительности пользовательского интерфейса фиксируют именно эти три параметра. Мы можем применять такие инструменты для непрерывного измерения этих показателей в конвейере CI и тем самым реализовать раннее тестирование производительности. Далее обсудим, как это сделать.

Упражнения

Из модели RAIL следует, что производительность пользовательского интерфейса определяет ощущения конечного пользователя. Чтобы измерить показатели производительности интерфейса вашего приложения, необходимо сначала определить набор тестовых сценариев, которые будут охватывать все впечатления пользователей из разных демографических групп. В частности:

- постарайтесь учесть многообразие устройств у пользователей (настольные компьютеры, мобильные телефоны, планшеты). Соберите информацию о производителях устройств, являющихся важными игроками в регионе, который будет обслуживать ваше приложение. Это важно, так как у каждого устройства есть свой процессор, аккумулятор и объем памяти, что влияет на удобство работы конечного пользователя;
- не забудьте про разную пропускную способность сетей у разных пользователей: Wi-Fi, 3G, 4G и т. д. Также имейте в виду, что средние скорости мобильной и широкополосной связи в разных странах различаются. Согласно данным World Population Review (<https://oreil.ly/iioOl>) на 2021 год, в Монако, например, была самая высокая средняя скорость широкополосного доступа в Интернет — 261,8 Мбит/с по сравнению с 203,8 Мбит/с в США, 102,2 Мбит/с в Великобритании, 81,95 Мбит/с в России и 13,8 Мбит/с в Пакистане;
- учитывайте распределение целевых пользователей. Как следует из предыдущего пункта (хотя есть и другие факторы, которые также вносят свой вклад), производительность пользовательского интерфейса, наблюдаемую в различных географических точках, следует тестировать специально.

В Интернете можно найти множество исследований, посвященных такого рода данным. Кроме того, если у вас уже есть действующее приложение, то Google Analytics предоставит вам информацию об использовании сайта в реальном времени. Получив тестовые сценарии, вы сможете применить описанные здесь инструменты для измерения показателей производительности пользовательского интерфейса, а также добавить эти тесты в свой конвейер CI.

Давайте для выполнения практических упражнений возьмем такой тестовый сценарий: «Пользователь из Милана, имеющий смартфон Samsung Galaxy S5, заходит на домашнюю страницу Amazon, используя сетевое соединение 4G». Теперь посмотрим, как различные инструменты, такие как WebPageTest и Lighthouse, могут помочь измерить производительность пользовательского интерфейса.

WebPageTest

WebPageTest (<https://www.webpagetest.org>) — бесплатный онлайн-инструмент для оценки производительности пользовательского интерфейса веб-сайта. Это мощный инструмент, включающий возможность выбора географической точки, откуда выполняется доступ к веб-сайту, и собирающий показатели производительности интерфейса путем отображения веб-сайта в реальных мобильных и настольных веб-браузерах. Едва ли вы найдете другой инструмент, настолько близко воспроизводящий поведение реального конечного пользователя, как этот!

Рабочий процесс

Этот инструмент очень прост в применении.

1. Введите URL домашней страницы Amazon в поле ввода (рис. 8.14).

The image shows the WebPageTest configuration page. At the top, there's a text input field containing 'https://www.amazon.com/' and a 'Start Test ->' button. Below this, the 'Test Location' is set to 'Milan, Italy - EC2 (Chrome,Firefox)' with a 'Select from Map' button. The 'Browser' is set to 'Samsung Galaxy S5'. Under the 'Advanced Settings' section, there are tabs for 'Test Settings', 'Advanced', 'Chromium', 'Script', 'Block', 'SPOF', and 'Custom'. The 'Test Settings' tab is selected, showing options for 'Connection' (4G (9 Mbps, 170ms RTT)), 'Desktop Browser Dimensions' (default (1366x768)), 'Number of Tests to Run Up to 9' (3), 'Repeat View' (radio buttons for 'First View and Repeat View' and 'First View Only'), and 'Capture Video' (checked checkbox).

Рис. 8.14. Настройки WebPageTest

2. Выберите местоположение конечного пользователя, тип браузера, тип мобильного устройства и пропускную способность сети в соответствии с тестовым примером.

3. В параметре **Number of Tests to Run** (Количество прогонов теста) установите значение 3. Результаты одного прогона могут быть ошибочными из-за сбоев в сети, поэтому рекомендуется запустить тестовый сценарий несколько раз и использовать среднее значение.
4. В группе **Repeat View** (Повторный просмотр) отметьте кнопку **First View and Repeat View** (Первый и повторный просмотр). В этом случае показатели эффективности будут фиксироваться отдельно для первого и последующих обращений к сайту. Как вы помните, они могут меняться для последующих посещений из-за кэширования.
5. Запустите тест и просмотрите отчеты с результатами.

Поскольку WebPageTest — это бесплатный общедоступный инструмент, вам, возможно, придется подождать в очереди несколько минут, чтобы просмотреть отчет. Чтобы избежать ожидания, можете настроить его в частном порядке в локальной тестовой среде за определенную плату.

В отчете имеется много разделов с ценной для отладки информацией. Каждый отчет можно получить по уникальному идентификатору в течение 30 дней. Давайте обсудим несколько важных разделов отчета, созданного в WebPageTest для нашего тестового сценария.

Таблица (рис. 8.15) содержит основные показатели производительности, такие как время до отрисовки первого содержимого (**First Contentful Paint**) во всех трех тестовых прогонах. Чтобы оценить время загрузки страницы, можно взять среднее время отображения документа (**Document Complete**) по всем трем прогонам. Обратите внимание на то, что время отображения документа при первом обращении составляет 3,134 с, а время до отрисовки наиболее заметного элемента содержимого (**Largest Contentful Paint**) — 2,105 с, то есть производительность взаимодействий с пользователем находится в допустимых пределах. Время полной загрузки (**Fully Loaded**) в таблице включает время, необходимое для загрузки всего вторичного контента, то есть задач, отложенных событием загрузки. Несмотря на существенную величину (~14 с при 230 запросах), оно вряд ли повлияет на работу конечного пользователя.

Performance Results (Median Run - SpeedIndex)													
	First Byte	Start Render	First Contentful Paint	Speed Index	Web Vitals			Document Complete			Fully Loaded		
					Largest Contentful Paint	Cumulative Layout Shift	Total Blocking Time	Time	Requests	Bytes In	Time	Requests	Bytes In
First View (Run 1)	0.918s	2.000s	1.994s	2.505s	2.105s	0.156	0.162s	3.134s	38	406 KB	14.615s	230	1,154 KB
Repeat View (Run 1)	1.156s	2.100s	2.085s	2.577s	2.316s	0.142	0.050s	3.048s	9	116 KB	13.502s	127	127 KB

Рис. 8.15. Таблица с показателями производительности из отчета WebPageTest

Каскадное представление, показанное на рис. 8.16, демонстрирует красочную шкалу распределения времени между задачами, такими как разрешение DNS, установка

соединения, загрузка HTML и изображений, время выполнения сценариев и т. д. Глядя на нее, можно понять, в каком направлении сосредоточить свои усилия для дальнейшей оптимизации.

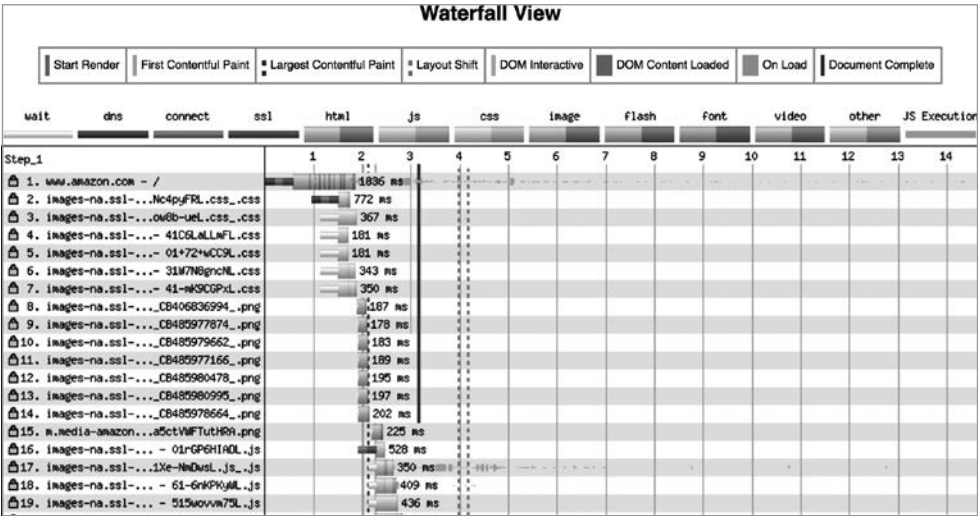


Рис. 8.16. Каскадное представление из отчета WebPageTest

WebPageTest имеет и средства для передачи данных аутентификации, но учтите, что предоставленные вами учетные данные смогут увидеть все, кто получит доступ к отчету с результатами тестирования.

WebPageTest также предоставляет API для программного создания отчетов, а кроме того, существует модуль для Node.js, позволяющий запускать тесты непосредственно из командной строки. Эти два варианта обеспечивают интеграцию с конвейерами CI. Оба они требуют ключа API, который можно получить за определенную плату. Если вы решите приобрести его, то за подробной информацией о командах CLI и использовании API обращайтесь к примерам 8.3 и 8.4 соответственно.

Пример 8.3. Команды WebPageTest CLI для установки модуля, запуска тестовых сценариев и просмотра результатов

```
// Шаг 1. Установка с помощью npm

npm install webpagetest -g

// Шаг 2. Запуск тестового сценария из командной строки

webpagetest test http://www.example.com --key API_KEY
--location ec2-eu-south-1:Chrome --connectivity 4G
```

```
--device Samsung Galaxy S5 --runs 3 --first --video  
--label "Using WebPageTest" --timeline
```

```
// Шаг 3. Прочитать результаты тестирования по идентификатору,  
// сгенерированному предыдущей командой
```

```
webpagetest results 2345678
```

Пример 8.4. WebPageTest API для запуска тестовых сценариев и просмотра результатов

```
// Шаг 1. Запуск тестового сценария с использованием  
// WebPageTest API
```

```
http://www.webpagetest.org/runtest.php?url=http%3A%2F%2Fwww.example.com&k=API_  
KEY&location=ec2-eu-south-1%3AChrome&
```

```
// Шаг 2. Прочитать результаты тестирования по идентификатору,  
// полученному в ответ на предыдущий запрос
```

```
http://www.webpagetest.org/jsonResult.php?test=2345678
```

Lighthouse

Lighthouse (<https://oreil.ly/rfWY0>) входит в состав Google Chrome, доступен также как расширение Firefox. Этот инструмент исследует указанный ему сайт по нескольким направлениям, включая безопасность, доступность и производительность интерфейса. Отчет Lighthouse о тестировании производительности включает общую оценку и все основные показатели.

Одно из преимуществ Lighthouse — он не размещается в публичном доступе, поэтому не требуется ждать, когда подойдет ваша очередь. Кроме того, он работает в вашем локальном браузере, поэтому не имеет проблем с безопасностью, правда, также это означает, что вы не сможете настроить географическую точку нахождения конечного пользователя (веб-сайт будет тестироваться из вашего фактического местоположения). Однако вы все еще можете регулировать параметры сети и процессора и изменять размеры окна браузера Chrome, чтобы симулировать различные тестовые сценарии и получить соответствующие показатели.

Lighthouse доступен и в виде инструмента командной строки, что упрощает его интеграцию в CI и позволяет получать непрерывную обратную связь. Zalando, ведущая европейская розничная сеть, заявила, что благодаря Lighthouse CI она сократила время получения обратной связи о производительности пользовательского интерфейса с одного дня до 15 минут (<https://web.dev/zalando>). Инструмент полностью бесплатный и имеет открытый исходный код.

Рабочий процесс

Чтобы опробовать Lighthouse, выполните следующие простые шаги.

1. Откройте сайт Amazon в Chrome.
2. Откройте Chrome DevTools, используя сочетание клавиш **Cmd+Option+J** в macOS, **Shift+Ctrl+J** в Windows/Linux или выбрав пункт **Inspect** (Просмотреть код) в контекстном меню Chrome.
3. Настройте условия работы сети на вкладке **Network** (Сеть). Например, выберите пункт **Slow 3G** (3G (низкая скорость)).
4. Настройте условия работы процессора на вкладке **Performance** (Производительность). Для имитации мобильных устройств со средним и низким быстродействием используйте замедление в четыре и шесть раз соответственно. Для примера выберите замедление в четыре раза.
5. Выберите в раскрывающемся списке размер окна. Например, укажите пункт **Galaxy S5** (рис. 8.17).
6. На вкладке **Lighthouse** выберите категорию **Performance** (Производительность) и нажмите кнопку **Generate report** (Анализ загрузки страницы).

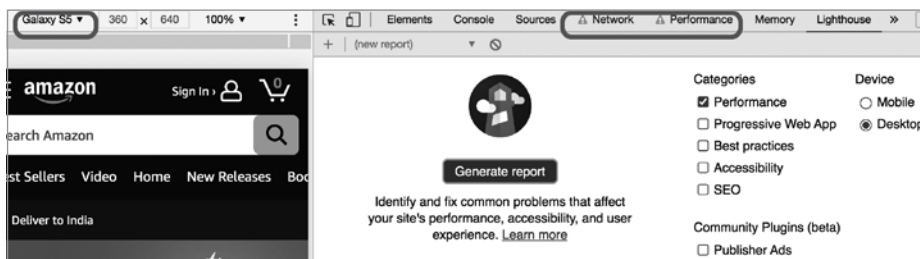


Рис. 8.17. Окно Lighthouse с настройками сети, процессора и разрешения экрана

Результаты, показанные на рис. 8.18, говорят о том, что Amazon неплохо справляется со своей задачей. Значение **Time to interactive** (Время до интерактивности) даже в таких жестких условиях составляет 3,8 с!

Lighthouse можно применять для тестирования производительности пользовательского интерфейса еще во время разработки. Для интеграции с CI можно задействовать модуль **Lighthouse Node.js**. Чтобы установить его, выполните в окне терминала следующую команду:

```
$ npm install -g lighthouse
```

Чтобы запустить тест производительности, выполните команду:

```
$ lighthouse https://www.example.com/ --only-categories=performance
```

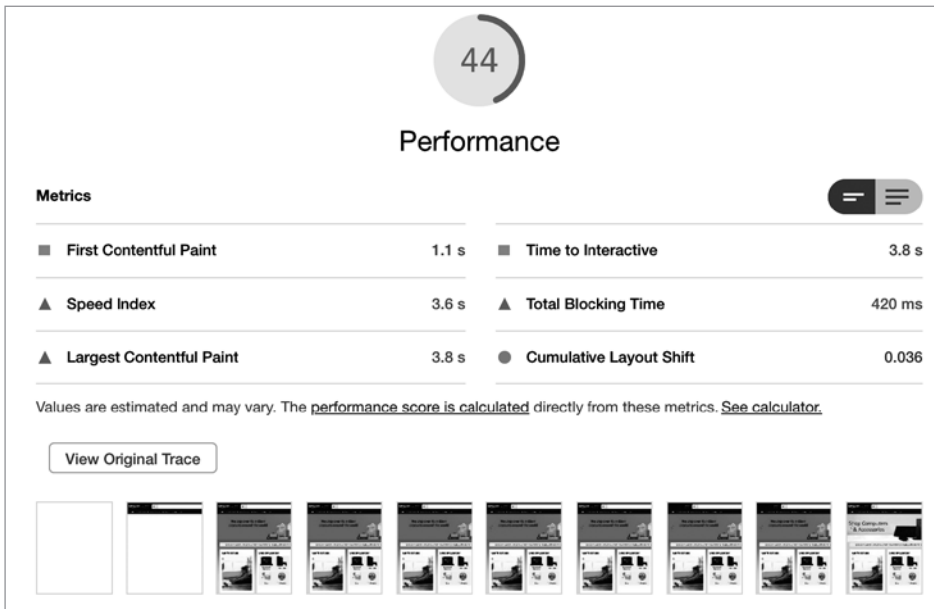


Рис. 8.18. Отчет о производительности, генерируемый инструментом Lighthouse

Ей можно передать дополнительные параметры (<https://github.com/GoogleChrome/lighthouse>), чтобы настроить характеристики работы сети и процессора и выбрать размеры экрана устройства. Отчет с результатами тестирования будет создан в текущем каталоге. Вы можете написать сценарий-обертку, который будет вызывать сбой конвейера, если показатель производительности окажется хуже порогового значения. Например, можно прервать сборку, если общая оценка будет меньше 90. Аналогично с помощью функции `LightWallet` (<https://oreil.ly/EefD9>) можно определить бюджеты производительности (верхние пороговые значения) для каждого веб-показателя. Это позволит сверить результаты производительности Lighthouse с определенными пороговыми значениями для каждого показателя и выдавать оповещения при их превышении.

Другой способ интеграции с CI — через инструмент `cypress-audit` (<https://oreil.ly/OwVi5>). Он интегрирует Lighthouse с Cypress и дает возможность тестировать производительность в ходе функционального тестирования в CI.

Дополнительные инструменты тестирования

Далее представлены еще несколько инструментов, которые помогают измерять и отлаживать производительность пользовательского интерфейса. В этом разделе мы рассмотрим некоторые функции PageSpeed Insights и Chrome DevTools.

PageSpeed Insights

Инструменты, которые мы задействовали в предыдущих упражнениях, позволяют выполнять тестирование как в лабораторных условиях, давая возможность настроить предварительные условия и наблюдать за результатами. Но в реальной жизни возникает множество нюансов из-за незначительных различий в пропускной способности сети, конфигурациях устройств и т. д., которые невозможно предсказать и измерить. Единственный способ узнать, как разные пользователи на самом деле оценивают производительность веб-сайта, — это мониторинг реальных пользователей (Real User Monitoring, RUM) после выпуска приложения. Google предоставляет бесплатные сервисы мониторинга, которые фиксируют основные показатели работы сети вместе с другими показателями, когда пользователи со всего мира обращаются к приложению. Эти данные называются *полевыми данными* (field data) или *RUM-данными* (RUM data).

Инструмент PageSpeed Insights пытается дать целостное представление о производительности пользовательского интерфейса, представляя RUM-данные вместе с лабораторными данными, полученными с помощью Lighthouse (рис. 8.19). Попробуйте поработать с этим инструментом, для чего введите URL действующего приложения на домашней странице PageSpeed Insights (<https://oreil.ly/NONIO>).

PageSpeed Insights предлагает также API для постоянного мониторинга и оповещений.

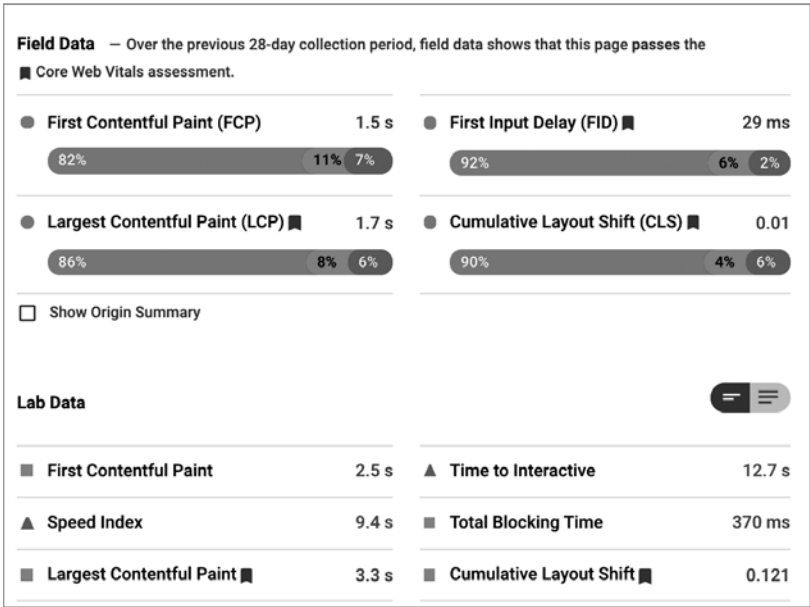


Рис. 8.19. Отчет с лабораторными и полевыми данными PageSpeed Insights

Chrome DevTools

Еще один удобный инструмент для отладки производительности пользовательского интерфейса — профилировщик производительности (<https://oreil.ly/Tkyqm>), доступный на вкладке **Performance** (Производительность) в Chrome DevTools. Он генерирует подробные аналитические отчеты о работе сети, частоте кадров анимации, потреблении графического процессора, памяти, времени выполнения сценариев и многом другом, что позволит разработчикам сэкономить драгоценные миллисекунды. Профилировщик также дает возможность настраивать характеристики работы сети и процессоры во время отладки. А будучи встроенным в сам браузер, он очень удобен для разработчиков.

Рассмотрим короткий пример. Предположим, вы хотите узнать, как работает автоматически заполняемый раскрывающийся список в пользовательском интерфейсе вашего приложения. Записать действие по вводу текста в раскрывающийся список можно, воспользовавшись кнопкой **Запись** на вкладке **Performance** (Производительность). Как только запись будет остановлена, на той же вкладке появится отчет об анализе производительности (рис. 8.20).

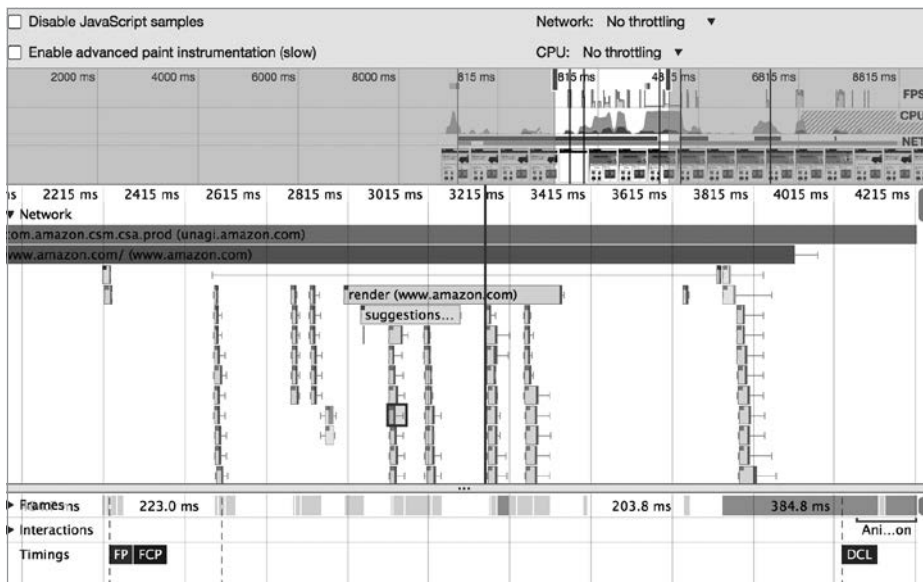


Рис. 8.20. Пример отчета профилировщика производительности в Chrome DevTools

При этом вы должны быть готовы приступить к сквозному тестированию производительности своего приложения. Последний шаг — собрать все это вместе и выработать стратегию тестирования производительности, чтобы заранее спланировать необходимые время и мощность.

Стратегия тестирования производительности

Как уже упоминалось в этой главе, руководящим принципом стратегии тестирования производительности должно стать раннее тестирование. Его следует начать с проектирования архитектуры (она должна соответствовать ожидаемым показателям производительности) и распространить на интеграцию тестов производительности в конвейеры CI для частой и постоянной обратной связи, чтобы напоминать, насколько это выгодно для бизнеса, благоприятно для конечных пользователей и ваших планов на выходные. На рис. 8.21 показан обзор стратегии тестирования производительности на ранних этапах разработки ПО, в которой применяются основы, обсуждаемые в этой главе.



Рис. 8.21. Стратегия тестирования производительности на ранних этапах разработки ПО

Давайте пройдемся по различным этапам тестирования производительности на ранних этапах разработки ПО. На этапе планирования:

- до начала проектирования согласуйте ключевые показатели эффективности (KPI) работы приложения со всеми заинтересованными сторонами, включая представителей бизнеса, маркетинга и технических специалистов. Используйте согласованные цифры при проектировании архитектуры и выборе технологического стека;

- в начале проекта настройте среду тестирования производительности. Если нельзя сделать ее похожей на промышленную среду, то хотя бы создайте среду, в которой можно начать тестирование;
- включите различные тесты производительности пользовательского интерфейса (например, состояние сети, географическое местоположение и т. д.) в критерии приемки каждой пользовательской истории;
- включите ожидаемые ключевые показатели эффективности API (время отклика, параллелизм и доступность) в критерии приемки каждой пользовательской истории.

В ходе разработки:

- проверяйте соответствующие KPI серверов (путем измерения времени отклика и проведения нагрузочного тестирования конечных точек) для каждой пользовательской истории;
- тестируйте производительность пользовательского интерфейса для каждой пользовательской истории.

В CI:

- запускайте все тесты, проверяющие время отклика для каждого коммита. В зависимости от времени, необходимого на выполнение нагрузочных тестов, запускайте их для каждого коммита или в ходе ночного регрессионного тестирования, чтобы как можно быстрее выявить проблемы с производительностью. Это поможет вам увидеть, как снижается производительность по мере добавления дополнительных функций, и поможет при последующей отладке;
- включите в свой CI тесты производительности для часто посещаемых страниц.

Во время тестирования пользовательской истории:

- следите за видимыми узкими местами производительности в ходе исследовательского тестирования;
- прежде чем отмечать пользовательскую историю как завершенную, убедитесь, что критерии приемки, связанные с производительностью, выполнены, автоматизированы и интегрированы в CI.

И наконец, на этапе тестирования релиза выполняйте сквозное тестирование производительности приложения, включая стресс-тестирование и тестирование на стабильность, а также действия по отладке. Перед этим этапом постарайтесь настроить среду тестирования производительности так, чтобы она была максимально похожа на промышленную.

Как вы уже наверняка поняли, тестирование производительности требует значительных усилий и не может быть внезапно включено в цикл выпуска позднее, когда вас посетит такая идея, не нарушая сроков сдачи проекта!

Ключевые выводы

- Плохая производительность может иметь серьезные финансовые последствия для бизнеса. И наоборот, повышение производительности способно значительно повысить доход.
- На производительность приложения влияют разные факторы, такие как архитектура, производительность сторонних сервисов, пропускная способность сети, географическое местоположение пользователя и т. д. Эти факторы постоянно меняются на протяжении всего цикла поставки программного обеспечения, и иногда оптимизацию одного показателя можно выполнить только за счет другого, что ставит разработчиков перед сложным выбором.
- Непрерывное измерение ключевых показателей эффективности (доступность, параллелизм/пропускная способность и время отклика) с начала цикла поставки программного обеспечения поможет предотвратить серьезные проблемы с производительностью в промышленной среде.
- Помочь реализовать раннее тестирование производительности могут несколько инструментов, таких как JMeter, Gatling и Apache Benchmark.
- Крайне важно уделять особое внимание производительности пользовательского интерфейса, потому что на выполнение кода пользовательского интерфейса приходится в среднем до 80 % времени загрузки приложения.
- Модель RAIL от Google дает прочную основу для определения показателей производительности фронтенда.
- Разрабатывайте сценарии тестирования производительности фронтенда с учетом опыта конечного пользователя. Применяйте различные характеристики пользовательского окружения, такие как пропускная способность сети, географическое местоположение и возможности устройства.
- Включите тесты производительности API и фронтенда в конвейер CI и избавьте свою команду от неприятных сюрпризов, обусловленных падением производительности!

Тестирование доступности

Доступность важна для некоторых
и полезна для всех.

Принципы доступности W3C

Интернет — важная часть нашей жизни: мы используем его для покупки товаров, общения с друзьями и семьей, приобретения новых навыков и чтобы быть в курсе мировых новостей. Я не могу себе представить, насколько труднее было бы пережить пандемию COVID-19 без возможности работать, контактировать и получать информацию, которую дает Интернет. Обеспечение доступности этого важного продукта для всех пользователей с постоянными, временными или ситуативными ограничениями называется *веб-доступностью*. К таким пользователям относятся люди с нарушениями зрения, пожилые, водители автомобилей и сталкивающиеся с другими проблемами при доступе к Интернету. Доступность — это разновидность удобства применения с точки зрения веб-разработки и инклюзивности в гуманитарном плане.

Основная цель обеспечения доступности состоит в том, чтобы позволить людям с ограниченными возможностями пользоваться услугами Интернета и улучшать качество своей жизни. Мне нравится слоган: «Важна для некоторых, полезна для всех», придуманный организацией W3C Web Accessibility Initiative (WAI) (<https://www.w3.org/WAI>). Он подчеркивает, насколько функции доступности веб-сайтов полезны для всех пользователей независимо от ограничений по здоровью или других препятствий. Например, все мы предпочитаем четкий, структурированный макет, в котором легко найти разные части страницы, а на сайте легко ориентироваться. Аналогично наличие простых и понятных сообщений об ошибках и инструкций — фундаментальная потребность всех пользователей, а приложения с голосовой поддержкой быстро распространяются из-за простоты, которую они обеспечивают в быстро меняющемся мире.



Аббревиатура W3C расшифровывается как World Wide Web Consortium (Консорциум Всемирной паутины). Это международное сообщество, возглавляемое Тимом Бернерсом-Ли, изобретателем Всемирной паутины, которое работает с организациями-членами и широко известными пользователями над разработкой стандартов для Интернета. Организация W3C WAI определила глобальные стандарты доступности веб-сайтов, которые мы обсудим в этой главе.

Сместив акцент с точки зрения бизнеса, можно сказать, что сообщество людей с ограниченными возможностями образует третью по величине экономику (<https://oreil.ly/eIRVf>) в смысле покупательной способности, поскольку каждый пятый житель мира имеет некоторые ограничения. Это позволяет экономически обосновать инвестиции в поддержку веб-доступности.

Более того, наличие доступной сети часто является юридическим требованием. Согласно Конвенции ООН о правах инвалидов (United Nations Convention on the Rights of Persons with Disabilities, UN CRPD), доступ к информационно-коммуникационным технологиям, включая Интернет, — это фундаментальное право человека (<https://oreil.ly/v0RiB>). Во многих странах теперь существует правовое регулирование (<https://oreil.ly/3t0NH>) доступности веб-сайтов, основанное на этом праве, и в последние годы наблюдается всплеск количества судебных исков (<https://oreil.ly/V9qld>) против компаний, нарушающих его. Первое дело было выиграно в 2017 году, когда человек с нарушением зрения подал в суд на Winn-Dixie, сеть супермаркетов в США, из-за того что сайт компании не поддерживал программы чтения с экрана (хотя впоследствии это решение было отменено). Итак, по всем этим и многим другим причинам команды разработчиков ПО и предприятия должны уделять более пристальное внимание функциям веб-доступности.

В этой главе вы познакомитесь с приемами и инструментами тестирования веб-доступности. Вашему вниманию будут представлены обзор персонажей пользователей с ограниченными возможностями, экосистема инструментов и технологий, внутренняя работа программ чтения с экрана, а также рекомендации по обеспечению доступности веб-сайтов, утвержденные многими правительствами по всему миру. Вы также познакомитесь со средами веб-разработки, поддерживающими специальные возможности, и стратегиями раннего тестирования доступности. Наконец, здесь вы найдете упражнения, представляющие инструменты автоматизированного тестирования доступности, которые сможете включить в свою стратегию непрерывного тестирования, чтобы дать команде возможность постоянно заботиться о доступности создаваемого веб-сайта!



Инструменты для тестирования доступности на мобильных устройствах описаны в главе 11.

Введение

Начнем со знакомства с типами пользователей с ограниченными возможностями и их конкретными потребностями. После этого рассмотрим экосистему доступности и руководящие принципы веб-доступности.

Персонажи пользователей с ограниченными возможностями

Напомню, что для обозначения подмножества пользователей с похожими характеристиками применяется термин «персонаж». Мы часто создаем персонажей пользователей в проектах программного обеспечения, чтобы понять их конкретные потребности и учесть эти потребности на всех этапах разработки, начиная с проектирования. На рис. 9.1 показан набор персонажей пользователей с ограниченными возможностями.



Рис. 9.1. Персонажи пользователей с ограниченными возможностями

Эти персонажи можно определить так.

- Мэтт, 30-летний бизнесмен, недавно сломал руку. Ему трудно пользоваться мышью, поэтому он хотел бы иметь возможность взаимодействовать с сайтом только с помощью клавиатуры.
- Хелен — 80-летняя учительница на пенсии, которая в последнее время испытывает сложности с восприятием цветов. Для удобства ей необходим более контрастный пользовательский интерфейс с хорошо различимыми элементами фона и переднего плана, такими как изображения, ссылки, кнопки и т. д. Это требование применимо и к пользователям с дальтонизмом.
- Эбби — подросток с когнитивными нарушениями. Ей требуется время, чтобы изучить что-то новое, поэтому ей нужен ясный веб-макет с правильными заголовками и согласованными панелями и элементами навигации. Фреду (на рисунке не показан), водителю грузовика, который, находясь за рулем, хочет найти ближайшую заправочную станцию, тоже необходимо четкое представление информации, чтобы он мог быстро принять решение.
- Конни — незрячий менеджер магазина. Для использования Интернета ему нужны голосовой доступ и поддержка преобразования текста в речь.
- У Лакшми есть ребенок, которого она носит с собой большую часть дня. Ей тоже нужен голосовой доступ, чтобы отправлять сообщения.

- Майя — специалист по программному обеспечению, у нее нарушена координация движений, поэтому для работы в Интернете ей нужны крупные текст, кнопки и элементы управления. Это требование предъявляют и пользователи с дислексией и плохим зрением.
- Филипп — глухой и любит готовить. Ему нужны субтитры, чтобы понять видеоролики с рецептами, которые он любит смотреть.
- Сяо — владелец розничного магазина, говорящий по-китайски. Он учит английский всего пару месяцев, поэтому для доступа в Интернет Сяо нужны простые инструкции и понятный контент, не содержащий жаргона или сложных слов и предложений. Пользователи с когнитивными проблемами и проблемами в обучении тоже получают пользу от этой функции.

В совокупности наши персонажи демонстрируют проблемы со зрением (полная или частичная потеря), слухом, когнитивными и мышечными проблемами, а также временные ограничения. Цель состоит в том, чтобы позволить им всем воспринимать веб-страницу, понимать ее и взаимодействовать с ней подобно любому другому пользователю.

Экосистема доступности

Чтобы создавать доступные веб-возможности, важно понимать всю экосистему доступности. Сюда входят различные инструменты и технологии (помимо веб-технологий), которые взаимодействуют между собой и используются для доставки контента пользователям с временными и постоянными ограничениями. Например, слепой пользователь, такой как Конни, применяет для взаимодействия с Интернетом преобразование текста в речь и средства голосового управления. Для этого задействуются технологии чтения текста и отдания голосовых команд. Некоторым другим людям необходимы другие вспомогательные устройства, интегрированные в компьютер. Итак, чтобы представить разные варианты использования людьми с ограниченными возможностями различных компонентов, мы должны иметь хотя бы самое общее представление о них. Далее перечислены элементы экосистемы доступности, которые мы должны учитывать.

Инструменты и приемы веб-разработки

Совершенно очевидно, что инструменты веб-разработки, такие как HTML, CSS и т. д., должны иметь необходимую поддержку доступности. Например, чтобы передать программе чтения с экрана информацию об имеющихся на странице элементах, в средах веб-разработки должны быть предусмотрены средства, позволяющие явно упоминать их.

Пользовательские агенты

Это инструменты, отображающие веб-контент, например браузеры и медиаплееры. Пользовательские агенты должны понимать, что веб-контент оснащен

функциями, связанными с доступностью, и интегрироваться с другими инструментами, такими как программы чтения с экрана.

Вспомогательные технологии

Вспомогательные технологии — это дополнительные устройства и технологии, которые взаимодействуют с браузером и передают информацию пользователю и от него, например, программы чтения с экрана, альтернативные клавиатуры, переключатели и т. д.

Как видите, экосистема доступности включает обширный набор инструментов и технологий. Поддержка доступности в этих компонентах позволяет всем пользователям взаимодействовать с Интернетом. Некоторые из инструментов и технологий могут предлагать более продвинутые возможности, чем другие, из-за чего в какой-то области может потребоваться выполнить больше обходных маневров из-за отсутствия некоторых функций поддержки наших персонажей пользователей.

Чтобы гарантировать наличие стандартных функций доступности во всех этих компонентах, W3C WAI определила следующие международные стандарты для каждого из них.

- *Рекомендации по обеспечению доступности средств разработки* (Authoring Tool Accessibility Guidelines, ATAG) определяют стандарты для инструментов создания контента, таких как редакторы HTML.
- *Рекомендации по обеспечению доступности веб-контента* (Web Content Accessibility Guidelines, WCAG) определяют стандарты веб-контента, на которые следует обращать внимание во время разработки.
- *Рекомендации по обеспечению доступности пользовательских агентов* (User Agent Accessibility Guidelines, UAAG) касаются стандартов для браузеров и медиаплееров, включая некоторые вспомогательные технологии.

Все эти стандарты подробно описаны на сайте WAI (<https://oreil.ly/Y9HzW>). Как веб-разработчики, в следующем разделе мы углубимся в знакомство с WCAG, в частности, WCAG 2.0, где говорится, какие характеристики должен иметь веб-контент (текст, изображения, цвета, мультимедиа и т. д.), чтобы считаться доступным. Как упоминалось ранее, многие страны выработали правовые нормы по применению стандарта WCAG 2.0 в государственном, общественном и частном секторе.

Пример. Программы чтения с экрана

Чтобы осознать, почему WCAG 2.0 дает те или иные рекомендации, необходимо понимать, как работают вспомогательные технологии. Рассмотрим для примера программы чтения с экрана, используемые людьми с ослабленным зрением, — это распространенная вспомогательная технология, поддержку которой мы обязательно должны протестировать.

Как следует из названия, программы чтения с экрана читают содержимое страницы вслух для пользователя, взаимодействующего с веб-сайтом с помощью клавиатуры, то есть, прослушивая контент, он может нажимать сочетания клавиш, такие как Tab, Tab+Shift, Enter и т. д., для взаимодействия с сайтом.

Программа чтения с экрана воспроизводит содержимое страницы в порядке, определяемом ее *деревом доступности* (accessibility tree). Это структура, подобная DOM, в которой элементы страницы вместе с атрибутами, такими как роли, идентификаторы и т. д., явно определяются в последовательности, представляющей значимый поток. Например, рассмотрим сайт бронирования с текстовыми полями ввода Куда и Откуда и кнопкой Искать на главной странице. Дерево доступности будет структурировано так, чтобы представлять поток поиска билетов, то есть сначала вводится местоположение «Откуда», затем местоположение «Куда», а затем нажимается кнопка Искать. При необходимости можно закодировать на веб-странице определенные элементы, чтобы они были скрыты в дереве доступности.

Чтобы лучше понять суть специальных возможностей доступности, желательно самим попробовать использовать программу чтения с экрана. Google Chrome предоставляет браузерную программу чтения с экрана в качестве расширения (<https://oreil.ly/S0Eie>). Попробуйте! Существуют также демонстрационные веб-сайты, такие как образец сайта бронирования (<https://oreil.ly/nxusv>) (рис. 9.2), помогающие понять, как люди с нарушениями зрения могут пользоваться Интернетом: контент намеренно размыт и вы можете выполнить бронирование, только с помощью программы чтения с экрана и клавиатуры.

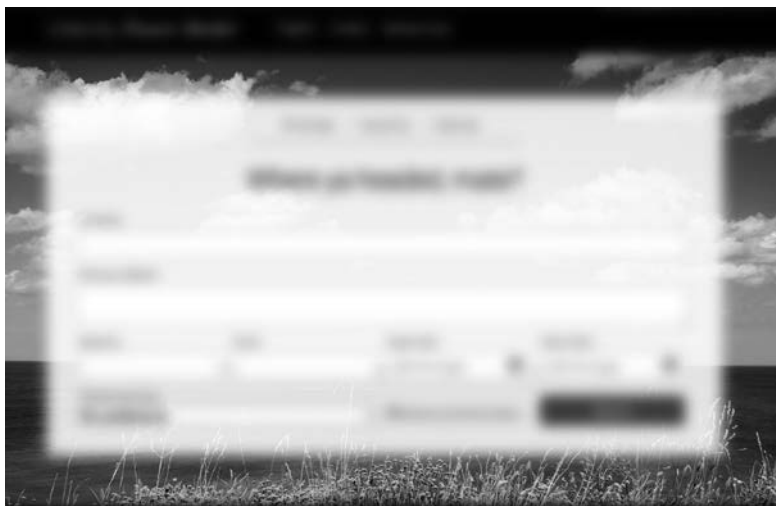


Рис. 9.2. Скриншот демонстрационного веб-сайта, содержимое которого намеренно размыто, чтобы побудить прибегнуть к помощи программы чтения с экрана

WCAG 2.0: руководящие принципы и уровни

Если у вас была возможность попробовать программу чтения с экрана, отлично! Если нет, то я надеюсь, вы получили представление о том, как все работает, прочитав предыдущий раздел. Теперь подробно рассмотрим рекомендации WCAG 2.0.

WCAG 2.0 определяет четыре руководящих принципа, о которых следует помнить при разработке веб-контента: контент должен быть *воспринимаемым, управляемым, понятным* и *надежным*. Стандарт определяет также три уровня соответствия критериям успеха, определенным для каждого из руководящих принципов.

Уровень А

Это минимальный уровень соответствия, без которого сайт будет считаться не соответствующим критериям доступности. Например, аудио- или видеоконтент должен иметь субтитры, все функции должны быть доступны с помощью клавиатуры, а цвет не должен служить единственным средством передачи информации. Соответствие уровня А позволит пользователям перемещаться по Сети.

Уровень АА

Сюда входят все требования уровня А плюс дополнительные более строгие требования, такие как минимальный уровень контрастности на сайте. Некоторые правовые нормы рекомендуют сайтам ориентироваться на этот уровень соответствия.

Уровень ААА

Этот уровень включает все требования двух предыдущих уровней плюс дополнительные расширенные требования, помогающие сделать сайт по-настоящему доступным для всех пользователей. Примером требования этого уровня может служить наличие сурдоперевода видеоконтента. Стремясь достичь такого уровня соответствия, вы показываете пользователям, что действительно заботитесь о них!

Организации должны определить уровень соответствия, основываясь на законодательных требованиях, но могут выбрать более высокий уровень соответствия для обслуживания дополнительных категорий пользователей.

Стандарты соответствия уровню А

Рассмотрим внимательнее требования WCAG 2.0 уровня А, которых должны придерживаться веб-сайты с минимальным уровнем соответствия. Их можно перенести в тесты доступности непосредственно.



Здесь дается только общий обзор некоторых деталей, чтобы вы могли понять суть тестирования требований к доступности. Желающие получить более полное представление о нем могут обратиться к официальной документации, имеющейся на сайте W3C WAI (<https://oreil.ly/k4cAv>).

Восприимчивость

Первый принцип — сделать веб-контент доступным для всех пользователей. Только имея возможность воспринимать контент, они смогут работать с ним. Итак, мы должны продумать все возможные сценарии, способные помешать выполнить это важное требование, еще на этапе проектирования и избегать их.

WCAG 2.0 определяет для этого принципа подробные требования.

- Весь нетекстовый контент, например изображения, должен сопровождаться альтернативным текстом, описывающим его, чтобы пользователи с нарушениями зрения могли понять, что изображено, с помощью программ чтения с экрана.
- Аудио- или видеоконтент должен иметь текстовые расшифровки и субтитры, синхронизированные с медиафайлами, а также позволять приостанавливать либо останавливать их и изменять громкость.
- Любой звук, который автоматически воспроизводится при загрузке страницы, должен иметь механизмы управления, такие как пауза, повтор и регулировка громкости.
- Информация и структура веб-страницы должны образовывать иерархическую структуру, например, заголовок страницы должен располагаться над всеми подзаголовками с соответствующими тегами и т. д. Это помогает пользователям программ чтения с экрана прослушивать содержимое страницы в осмысленном порядке.
- Инструкции по навигации не должны основываться исключительно на сенсорных характеристиках компонентов, таких как форма, цвет, размер, визуальное расположение, ориентация или звук. Например, избегайте инструкций, в которых говорится: «Подождите, пока кнопка не станет зеленой» или «Подождите, пока не услышите звуковой сигнал».
- Цвета не должны быть единственным способом обозначения действий, необходимости дать ответ или различения элементов на экране. Все это должно дублироваться с помощью текста, чтобы поддержать пользователей, страдающих проблемами с цветовосприятием.
- Фон страницы и элементы переднего плана должны иметь хороший цветовой контраст, чтобы помочь пользователям со слабой чувствительностью к цвету. Для этого существует фиксированное соотношение.

Управляемость

Обеспечив восприимчивость веб-контента, мы должны рассмотреть способы, позволяющие пользователям комфортно управлять веб-сайтом, например нажимать кнопки с помощью сочетаний клавиш. WCAG 2.0 включает следующие требования к управляемости.

- Должна существовать возможность навигации по сайту и управления им только с использованием клавиатуры. При навигации с помощью клавиатуры фокус на элементах должен четко фиксироваться и иметь соответствующий цветовой контраст.
- Должны иметься средства для перемещения вперед, назад и выхода с помощью сочетаний клавиш, например, должно поддерживаться сочетание клавиш для выхода из модального окна.
- У пользователей должно быть достаточно времени, чтобы полностью прочитать контент.
- Следует избегать контента, который мигает на экране и содержит множество анимаций, так как это может вызвать такие физиологические реакции, как судороги.
- Пользователи должны иметь возможность пропускать повторяющийся контент.
- Нужно скрывать закадровый контент от программ чтения с экрана. Например, если ссылка отображается только в определенном фрагменте, скройте ее в потоке чтения с экрана.
- Ссылки должны включать подробный и содержательный текст.

Понятность

Веб-сайту может потребоваться множество элементов и путей для выполнения действия. Например, чтобы забронировать авиабилет, может потребоваться выполнить несколько шагов и инструкций. Такой контент и пользовательские сценарии должны быть тщательно продуманными, простыми и понятными для всех пользователей. В отношении понятности WCAG 2.0 выдвигает довольно жесткие требования.

- Не применяйте жаргон и технические термины, давайте простое и содержательное описание. Например, избегайте сообщений о технических ошибках, таких как «034506451988 — недействительное число», текст должен быть понятным, например: «Неверный формат даты».
- При необходимости приводите расшифровку сокращений.
- Избегайте внезапных изменений контекста, например открытия нескольких окон, потому что это влияет на навигацию с помощью клавиатуры.

- Избегайте изменений контекста, когда у пользователя другие настройки, например более крупный шрифт.
- Сопровождайте элементы четкими и понятными подписями, чтобы помочь пользователям правильно выполнять действия. Например, поле ввода адреса электронной почты должно иметь метку «Электронная почта» и образец значения, например `example@xyz.com`.

Надежность

Наконец, веб-контент должен быть надежным и поддерживать различные типы пользовательских агентов и вспомогательных технологий. Программы чтения с экрана не единственные вспомогательные технологии, помимо них, существует множество других, требующих поддержки! WCAG 2.0 предъявляет следующие требования к этому принципу.

- Код на языке разметки должен соответствовать таким стандартам, как наличие открывающих и закрывающих тегов, отсутствие дубликатов, уникальные идентификаторы и т. д., чтобы его можно было легко анализировать с помощью вспомогательных технологий.
- Имя, роль и состояние каждого элемента, в том числе созданного сценариями, должны быть доступны для вспомогательных технологий (например, `role="checkbox"` и `aria-checked="true|false"`). Предоставляйте обновленное состояние элементов, таких как флажки, программе чтения с экрана после их выбора.

ВЕБ-ПРИЛОЖЕНИЯ, СООТВЕТСТВУЮЩИЕ WAI (WAI-ARIA)

Ранее мы видели, что программа чтения с экрана считывает элементы страницы и описывает действия, которые можно выполнить с ними, с помощью дерева доступности на веб-странице. Иногда, когда пользовательские элементы разрабатываются для расширенного взаимодействия с пользователем, вспомогательные технологии могут не идентифицировать их. Такие элементы должны иметь дополнительные атрибуты (<https://oreil.ly/hBj6R>), определяющие их тип, состояние и поведение, чтобы вспомогательные технологии могли их понять. Например, стандартное определение HTML-элемента, скажем `<input type="checkbox">`, будет автоматически идентифицировано как флажок (checkbox) и конечному пользователю будет правильно предложено выполнить щелчок на нем. Однако если элементу маркированного списка (``) придать вид флажка с помощью CSS, то его необходимо дополнить новыми атрибутами, чтобы вспомогательные технологии могли правильно его идентифицировать.

Документ WAI-ARIA (<https://oreil.ly/cSH9c>) — WAI's Accessible Rich Internet Applications («Веб-приложения, соответствующие требованиям доступности WAI») определяет спецификации для этих атрибутов (например, `roles`, `aria-checked` и т. д.), которые необходимо соблюдать при разработке веб-приложений. Эти атрибуты ARIA добавляются в дерево доступности, что делает его удобным для всех вспомогательных технологий.

Это основные требования уровня А. Существует также обновленная версия стандарта WCAG 2.1, которая включает еще несколько требований для поддержки определенных групп пользователей. Вы сможете изучить их в официальном документе (<https://oreil.ly/5LUcS>), если ваша организация решит соблюдать эту версию.

Фреймворки с поддержкой специальных возможностей

Для создания упомянутых ранее возможностей многие фреймворки предоставляют развитую поддержку специальных возможностей. Например, React полностью поддерживает создание доступных веб-сайтов, часто с помощью стандартных методов HTML. Аналогично команда Angular предлагает библиотеку Angular Material, содержащую набор повторно применяемых компонентов пользовательского интерфейса с поддержкой доступности. Vue.js тоже поддерживает создание доступных компонентов. Кроме того, существуют инструменты автоматизированного тестирования доступности (с ними вы познакомитесь в следующем разделе), предупреждающие, если в HTML отсутствуют стандартные теги, связанные с доступностью. Поэтому не волнуйтесь: вам обеспечена вся необходимая поддержка! И ваша команда может реализовать все необходимое без дополнительных усилий.

Стратегия тестирования доступности

Из предыдущих разделов должно быть очевидно, что нужно продумывать большинство требований к доступности с самого начала проекта и постоянно поддерживать их на протяжении всего процесса разработки, а не добавляться после этапа тестирования. Например, чтобы соответствовать уровню А, следует включить простую и последовательную навигацию по сайту, субтитры к видео, содержательные сообщения об ошибках, контрастные по цвету изображения и так далее уже на этапе проектирования, а не во время разработки или тестирования. В качестве первого шага по поддержке специальных возможностей командам требуется определить типы пользователей с ограниченными возможностями, как сказано в начале главы, и адаптировать пользовательские истории под особенности каждого из этих типов. Затем, когда команда обсудит функции продукта, нужно коллективно проверить, включены ли потоки доступности. Это будет первый шаг в реализации раннего тестирования доступности!

На рис. 9.3 показана реализация раннего тестирования доступности на протяжении всего жизненного цикла разработки ПО. Некоторые из этих вопросов мы подробно рассмотрим в оставшейся части раздела и в последующих упражнениях.



Рис. 9.3. Стратегия тестирования доступности на ранних этапах разработки ПО

Чек-лист доступности в пользовательских историях

Рекомендации WCAG 2.0 включают несколько универсальных требований, которые охватывают все страницы веб-сайта, например: наличие альтернативного текста, поддержка навигации с помощью клавиатуры, иерархическая организация заголовков страниц и т. д. Следовательно, добавление чек-листа доступности ко всем пользовательским историям поможет разработчикам и тестировщикам соблюдать эти требования и проверять их выполнение. Далее приводится общий чек-лист, который вы можете использовать в своей команде после добавления в приложение любых элементов.

ЧЕК-ЛИСТ ДОСТУПНОСТИ

- Проверьте заголовок страницы в браузере. При наведении указателя мыши на вкладку браузера в Chrome вы можете увидеть заголовок страницы в виде небольшого виджета. Этот текст должен четко определять контекст страницы.
- Проверьте базовую структуру веб-страницы и убедитесь, что она имеет правильные атрибуты элементов и иерархию. Отключите CSS и проверьте, все ли элементы расположены в порядке, удобном для чтения с экрана. В Chrome DevTools можно увидеть дерево доступности, отражающее порядок элементов.

- Проверьте возможность навигации исключительно с помощью клавиатуры, включая правильную подсветку элементов и комбинации клавиш для перехода назад, вперед и выхода.
- Убедитесь, что сообщения об ошибках, подписи, ссылки и вообще любой текст на странице такой, каким он задумывался.
- Проверьте удобочитаемость страницы после изменения размера текста, используя системные настройки или функцию масштабирования в браузере.
- Проверьте читабельность в оттенках серого. Например, пользователи Mac могут включить этот режим так: System Preferences ▶ Accessibility ▶ Display ▶ Use grayscale (Системные настройки ▶ Специальные возможности ▶ Отображение ▶ Использовать оттенки серого).
- Проверьте наличие субтитров к видео- и аудиоконтенту, убедитесь в их осмысленности и синхронизированности.
- Проверьте наличие содержательных альтернативных текстовых описаний для изображений. Для этого можно отключить загрузку изображений в настройках браузера (например, в Chrome выберите Settings ▶ Site Settings ▶ Images ▶ Block (Настройки ▶ Настройки сайта ▶ Картинки ▶ Запретить), после чего вместо изображений браузер будет отображать замещающий текст.
- Убедитесь, что поток чтения с экрана имеет смысл и что конечный пользователь сможет выполнить необходимые ему действия, следуя за этим потоком.

Инструменты автоматизированного тестирования доступности

Чек-лист доступности включает некоторые аспекты, которые можно проверить только с участием человека, например: просмотр сайта в оттенках серого, увеличение и уменьшение масштаба, проверка осмысленности альтернативного текста и т. д. Вы можете дополнить его инструментами автоматизированного тестирования, которые будут сканировать базовую структуру HTML и предупреждать, если у каких-то элементов отсутствуют теги доступности. Эти инструменты помогут вам сэкономить время и силы, предоставляя мгновенную информацию об отсутствующих тегах уже на этапе разработки.

Они представлены в виде средств статического анализа кода и проверки доступности во время выполнения. `eslint-plugin-jsx-a11y` (<https://oreil.ly/xsw0H>) — инструмент проверки для React, плагин ESLint, который проверяет соблюдение нескольких стандартов доступности непосредственно в вашем JSX. Аналогично `Codelyzer` (<https://oreil.ly/xjLFv>) имеет правила проверки стандартов доступности в исходном коде TypeScript, HTML, CSS и Angular. Эти инструменты будут давать обратную связь в ходе разработки, а средства проверки времени выполнения (такие как `axe-core`, `Pa11y CI` и `Lighthouse CI`, которые обсуждаются далее в этой главе) — обратную связь о фактическом положении дел после завершения разработки веб-страницы. Их можно запускать на локальной машине разработчика, чтобы быстрее получать информацию о созданных страницах как часть каждой пользовательской истории, а также как часть CI для непрерывного тестирования.

В дополнение к применению подобных инструментов можно выполнять тестирование на соответствие потребностям доступности, например наличие отдельного раздела расшифровки под любым видео/аудио или набора значимых сообщений об ошибках и инструкций, в форме автоматизированных функциональных тестов на микро- и макроуровне.

Ручное тестирование

Ручное тестирование имеет решающее значение для проверки доступности веб-сайта. Как упоминалось ранее, автоматизированные инструменты проверяют лишь структуру HTML, а чек-лист включает только обязательные элементы, общие для всех страниц. Помимо этого, в рамках ручного тестирования необходимо проверить множество элементов, например, пользовательского сценария с применением программы чтения с экрана и клавиатуры. Таким образом, в рамках ручного тестирования вы можете сосредоточиться на разных областях и разных этапах, например таких.

Тестирование пользовательских историй

В ходе тестирования пользовательских историй убедитесь, что все страницы, на которые распространяется пользовательская история, соответствуют чек-листу. Вы можете объединить эту проверку с инструментом оценки веб-доступности WAVE, бесплатным онлайн-сервисом от WebAIM. Он помогает обнаружить на веб-странице несоответствия рекомендациям WCAG 2.0. Даже притом что этот инструмент проверяет не все требования стандарта WCAG 2.0, он визуально выделяет проблемы на веб-странице в браузере, что может помочь вам заметить некоторые недостатки, о которых вы, возможно, не догадывались. В качестве альтернативы для тестирования доступности веб-страницы можно использовать Lighthouse, который, как мы видели в предыдущей главе, является частью Chrome DevTools. Оба эти инструмента обсуждаются далее в этой главе.

Тестирование функций

При тестировании на уровне пользовательских историй большая часть усилий по тестированию доступности ложится на ваши плечи. Но когда функция завершена, необходимо провести еще один раунд ручного тестирования, чтобы убедиться, что конечные пользователи способны выполнить свою работу, задействуя только клавиатуру, и что программы чтения с экрана могут перемещаться по функциям. Этот уровень тестирования функций поможет выявить любые недостатки согласованности в сквозной навигации приложения.

Чтобы протестировать навигацию с помощью одной лишь клавиатуры, используйте клавиши **Tab** и **Tab+Shift** для перемещения вперед и назад по веб-сайту, клавишу **Enter** — для выбора и клавиши со стрелками вверх и вниз — для

выбора элементов в раскрывающемся списке. При этом убедитесь, что фокус находится на правильном элементе и этот элемент четко выделен. Для тестирования процесса чтения с экрана можно использовать расширение Chrome, упомянутое ранее.

Выполнив эти проверки, вы завершите сквозное тестирование доступности функций.

Операционное тестирование

Наконец, когда все функции будут реализованы, рекомендуется провести тестирование с реальными конечными пользователями, включая людей с ограниченными возможностями, поскольку разные люди могут пользоваться различными вспомогательными устройствами. Это даст вам обратную связь в режиме реального времени перед сертификацией продукта на соответствие. UserTesting.com (<https://www.usertesting.com>) — это сервис удаленного тестирования, где вы сможете пригласить людей с ограниченными возможностями в качестве тестировщиков вашего сайта.

Сертификация соответствия

Когда веб-сайт готов, эксперты по стандартам WCAG проводят оценку соответствия, прежде чем он будет запущен. Это не централизованное подразделение — организации могут иметь собственных экспертов или нанимать консультантов для выполнения окончательной оценки, когда продукт будет готов к сертификации.

Поскольку каждый отдельный элемент на странице требуется изменить, чтобы сделать его доступным, раннее тестирование — единственный способ избавить вашу команду от необходимости решать сложную задачу по исправлению всех проблем доступности в конце цикла разработки.

Упражнения

Ранее я упомянула несколько инструментов автоматизированного тестирования доступности. Вы можете попробовать применить некоторые из них в следующих упражнениях.



Инструменты тестирования доступности помогают подтвердить целостность структуры HTML и предупреждают, если это не так. Например, они проверяют, все ли HTML-теги закрыты, все ли изображения имеют атрибут с альтернативным описанием, все ли элементы формы имеют подписи, все ли идентификаторы элементов уникальны и т. д. Они удобны для предварительной оценки и обеспечения быстрой обратной связи, но не устраняют необходимость ручного тестирования.

WAVE

WAVE — это онлайн-инструмент оценки доступности, который можно использовать для проверки веб-страницы на соответствие стандартам доступности. В нем есть возможности для проверки структуры страницы без помощи CSS и выявления таких проблем, как недостаточный цветовой контраст элементов, отсутствие атрибутов языка и т. д. WAVE прост и может использоваться бесплатно.

Рабочий процесс

Чтобы запустить тестирование с помощью инструмента WAVE, сделайте следующее.

1. Откройте веб-сайт WAVE (<https://wave.webaim.org>).
2. Введите в поле **Web page address** (Адрес веб-страницы) URL вашего приложения. В качестве альтернативы можно использовать демонстрационный веб-сайт WAI (<https://oreil.ly/qwuer>), который в учебных целях намеренно реализован с нарушениями требований доступности.
3. Щелкните на изображении стрелки, чтобы запустить тестирование.

На рис. 9.4 показаны результаты тестирования демонстрационного сайта WAI. Инструмент выявил три структурных элемента и шесть функций, а также 37 ошибок и предупреждений и две ошибки цветового контраста. Рядом с соответствующими элементами можно увидеть значки, сигнализирующие о неудачах, успехах и предупреждениях.

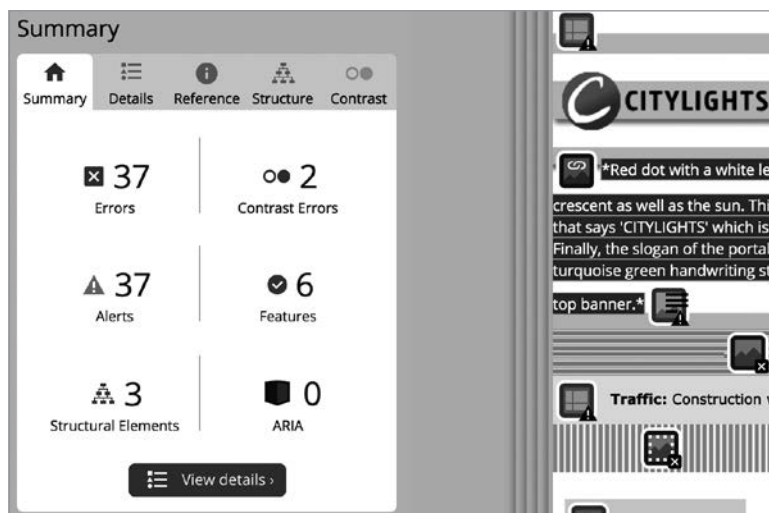


Рис. 9.4. Отчет инструмента WAVE с результатами тестирования демонстрационного сайта WAI

Если щелкнуть на вкладке **Details** (Подробности) рядом с надписью **Summary** (Сводка), то отобразятся сведения об ошибке (рис. 9.5).

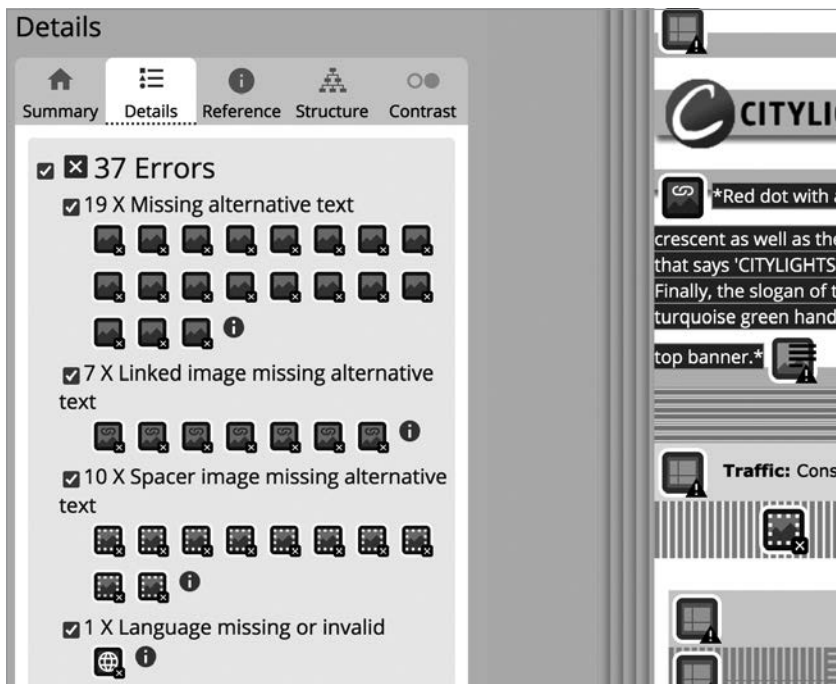


Рис. 9.5. Подробное описание ошибок в WAVE

Демонстрационная страница содержит 19 изображений без альтернативного текста, семь изображений, которые являются ссылками без альтернативного текста, десять изображений-разделителей без альтернативного текста и один отсутствующий или недопустимый атрибут языка. Для упрощения идентификации и отладки различные стили значков на вкладке **Details** (Подробности) можно сопоставить со значками на веб-странице.

Далее, чтобы увидеть структуру страницы, можно отключить стили CSS, используя элемент управления, расположенный над разделом сводки. На вкладке **Structure** (Структура) будут показаны результаты анализа структуры страницы. Как видно на рис. 9.6, когда стили отключены, блоки текста накладываются друг на друга и страница выглядит неряшливо. На веб-странице отсутствует правильная иерархия и нет разделов **Header**, **Navigation** и **Main**, что нарушает требования доступности.

Теперь попробуйте протестировать версию того же демонстрационного сайта WAI (<https://oreil.ly/EEMv7>), но реализованного с учетом требований доступности. Как видно на рис. 9.7, страница имеет правильно организованную структуру с иерархией.

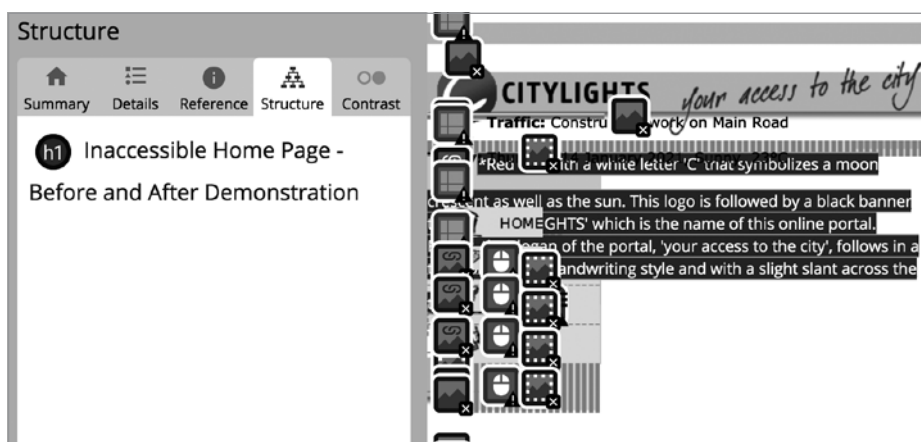


Рис. 9.6. Результаты анализа структуры страницы в WAVE с отключенными стилями

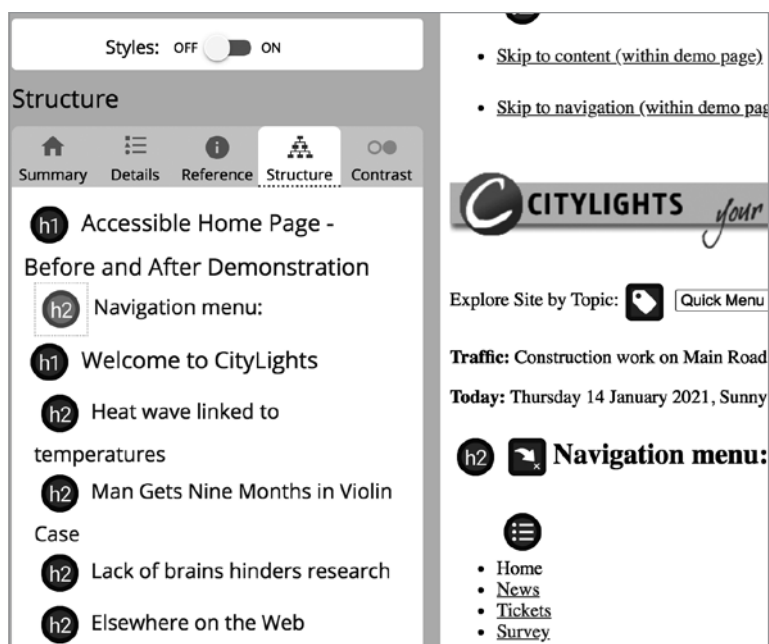


Рис. 9.7. Результаты анализа структуры страницы в WAVE на демонстрационном сайте, реализованном с учетом требований доступности

На сайте, реализующем поддержку специальных возможностей, можно проверить иерархическую организацию элементов и контента, а также убедиться, что навигация соответствует ожиданиям с учетом отображаемой иерархии.

Lighthouse

Если ваше приложение не общедоступное, то у вас может не быть возможности применить WAVE. В таких случаях можно воспользоваться инструментом Lighthouse от Google, который позволяет проверять доступность веб-сайта с помощью локального браузера Chrome.

Рабочий процесс

Выполните следующие простые шаги, чтобы увидеть, как работает Lighthouse.

1. Откройте в Chrome демонстрационный сайт WAI (<https://oreil.ly/qwuwer>), реализованный с нарушениями требований доступности.
2. Откройте Chrome DevTools, нажав комбинацию клавиш `Cmd+Option+I` в macOS или `Shift+Ctrl+J` в Windows/Linux.
3. На вкладке Lighthouse выберите категорию Accessibility (Доступность) (рис. 9.8) и нажмите кнопку Generate report (Анализ загрузки страницы).

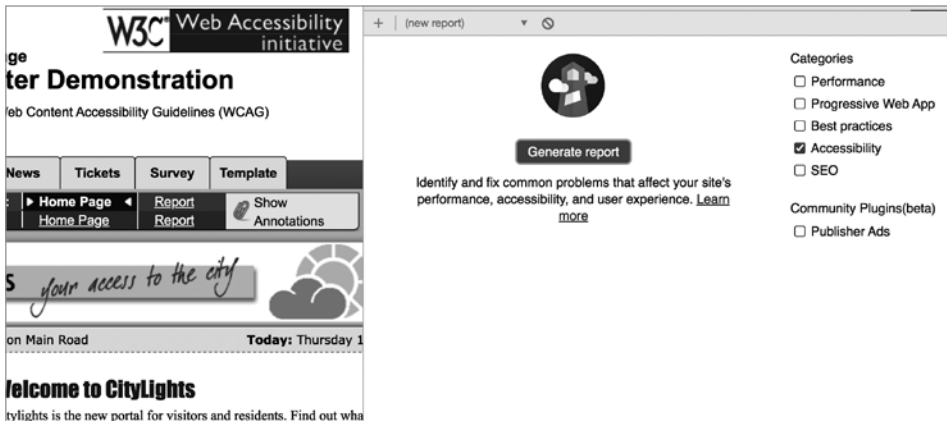


Рис. 9.8. Использование Lighthouse в Chrome DevTools для создания отчета о соответствии требованиям доступности

Отчет Lighthouse о тестировании доступности вскоре появится на той же панели (рис. 9.9).

Как видно на рис. 9.9, Lighthouse сообщает о проблемах, аналогичных тем, которые обнаружил WAVE (две проблемы с контрастностью, отсутствие атрибута `lang` и т. д.). Кроме того, для облегчения отладки представлены реальные строки кода, содержащие ошибки, и ссылки на рекомендации, которые помогут разработчикам их исправить. В верхней части отчета сообщается общая оценка, дающая

представление о том, насколько хороша или плоха страница, а в нижней части приводится чек-лист *Additional items to manually check* (Дополнительные элементы для проверки вручную), чтобы показать, чего тест не обнаружил. В этом списке тоже приводятся ссылки на рекомендации, которые помогут выполнить проверку вручную.

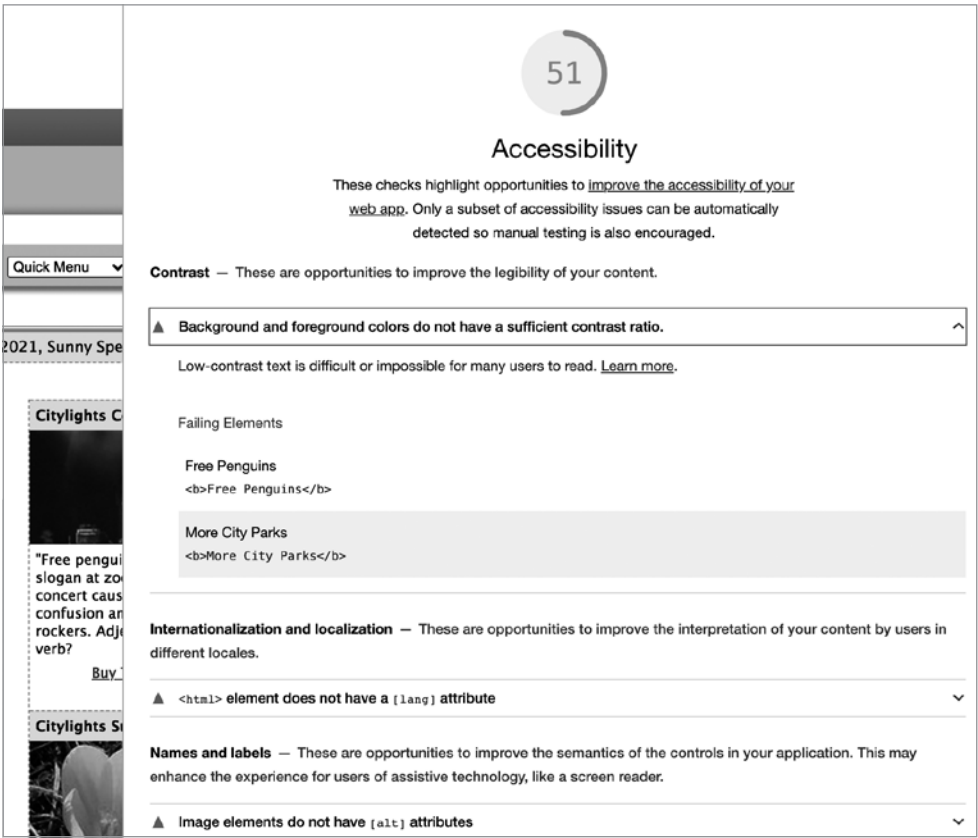


Рис. 9.9. Отчет Lighthouse с результатами тестирования демонстрационного сайта WAI, реализованного без учета требований к доступности

Модуль Lighthouse для Node

Модуль Lighthouse для Node (<https://oreil.ly/NkblQ>) выполняет тесты, похожие на тесты в версии, входящей в состав Chrome DevTools, но его можно запустить из командной строки. Благодаря этому его можно использовать для интеграции с CI, и он обеспечивает большую гибкость при настройке и составлении отчетов.

Рабочий процесс

Чтобы запустить тестирование доступности из командной строки, выполните следующие действия.

1. Чтобы установить модуль Lighthouse Node, если у вас уже установлена среда Node.js (<https://nodejs.org/en/download>), выполните команду:

```
$ npm i -g lighthouse
```

2. Запустите тестирование, выполнив команду:

```
$ lighthouse --chrome-flags="--headless" URL
```

По умолчанию отчет создается в виде HTML-файла, пример которого показан на рис. 9.10, в том же рабочем каталоге.

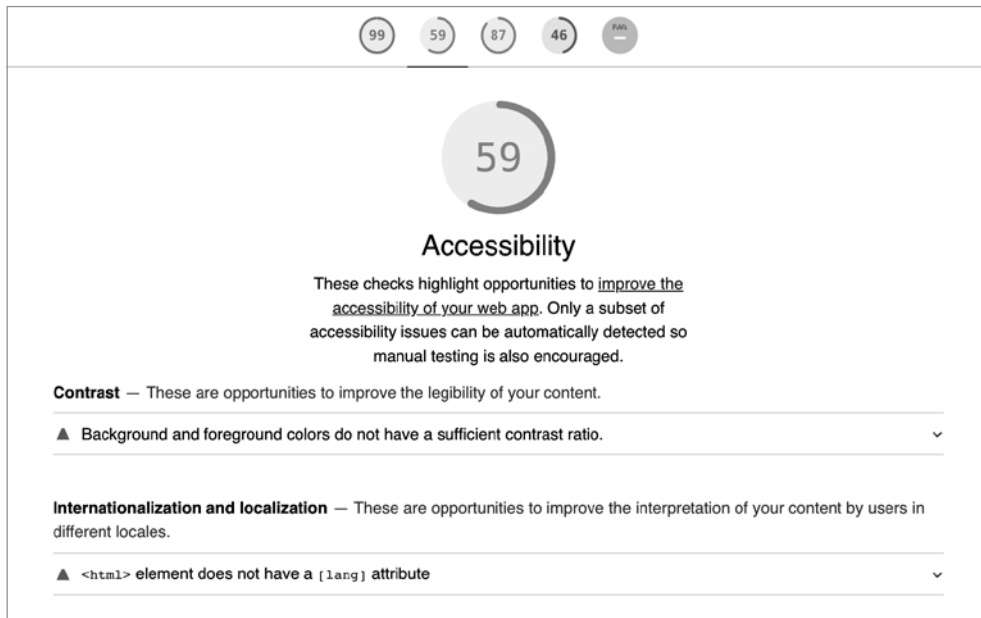


Рис. 9.10. Отчет модуля Lighthouse Node с результатами тестирования демонстрационного сайта WAI, реализованного без учета требований доступности

Для упрощения проверки можно добавить этот HTML-файл в число выходных артефактов конвейера CI. Кроме того, с помощью обертки для задачи сборки можно прервать процесс сборки, если оценка окажется ниже порогового значения.

Дополнительные инструменты тестирования

Еще несколько инструментов аудита, которые часто используются для тестирования доступности, — это Pa11y CI и axe-core. Посмотрим, что они предлагают.

Модуль Pa11y CI для Node

Pa11y CI (<https://github.com/pa11y/pa11y-ci>) — это инструмент командной строки, который поставляется в виде модуля Node. Он тестирует доступность одного или нескольких URL и сообщает о проблемах, подобно инструментам WAVE и Lighthouse. Чтобы с помощью этого инструмента протестировать несколько веб-страниц, добавьте их URL в раздел `urls` в конфигурационном файле, как показано в примере 9.1. Альтернативно можно передать параметр командной строки `--sitemap` с XML-картой сайта. Модуль Pa11y CI можно добавить в этап сборки проекта в CI или даже в отдельный этап.



Доступность (accessibility) иногда сокращается до a11y, что означает «a-[11 букв между ними]-y».

Пример 9.1. Конфигурационный файл Pa11y CI с перечисленными в нем URL демонстрационных сайтов WAI

```
{
  "defaults": {
    "timeout": 1000,
    "viewport": {
      "width": 320,
      "height": 480
    }
  },
  "urls": [
    "https://www.w3.org/WAI/demos/bad/after/home.html",
    "https://www.w3.org/WAI/demos/bad/before/home.html"
  ]
}
```

В инструменте предусмотрены параметры, позволяющие задавать пороговые количества ошибок и предупреждений, до которых может дойти сборка в CI, определять размеры экранов для тестирования и время ожидания загрузки страницы.

axe-core

В документации на GitHub (<https://oreil.ly/Whu2x>) для axe-core указано, что этот инструмент может автоматически обнаруживать в среднем 57 % проблем WCAG. Он работает со многими браузерами, включая Microsoft Edge, Google Chrome,

Firefox, Safari и IE. На основе этого инструмента построено множество расширений. Например, для интеграции с Java Selenium WebDriver можно добавить axe-core как зависимость Maven. Аналогично для интеграции с Cypress можно использовать модуль `cypress-axe` для Node. Есть также библиотеки `vue-axe` и `response-axe` для добавления тестов внешнего интерфейса.

По сути, axe-core предлагает API-функции для тестирования доступности веб-страниц, которые можно использовать в функциональных тестах. Например, функция `run()` запускает тестирование доступности текущей страницы и генерирует ошибки в случае сбоя. Проще говоря, вызов этой функции можно добавить в функциональные тесты для оценки доступности страницы.

Итак, мы познакомились с автономными инструментами, которые можно запускать из сценариев сборки проекта или в рамках отдельного этапа CI, а также с инструментами, интегрирующимися с функциональными тестами. Какой бы вариант вы ни выбрали, старайтесь разрабатывать стратегию непрерывного тестирования так, чтобы ваша команда получала обратную связь на как можно более ранних стадиях цикла разработки.

Затраченные вами время и силы обязательно окупятся, потому что вы сможете обеспечить удобный доступ к приложению Мэтту, Фреду, Хелен, Лакшми, Конни, Сяю, Эбби, Майе, Филиппу, а также себе самим!

Перспективы: доступность как культура

В этой главе мы обсуждали доступность в контексте веб-приложений, однако эти концепции применимы не только к веб-сайтам. Доступность — это культура, требующая изменения мышления. Приняв этот образ мышления, мы начинаем задавать себе такие вопросы: если я отправлю электронное письмо только с изображениями, без поясняющего текста, будет ли оно доступно для всех? Если я использую мелкий шрифт на слайдах презентации, смогут ли все их прочитать? Не лучше ли вместо специфического жаргона употреблять простые и ясные сообщения, чтобы все могли их понять? Я уверена, что все мы знаем ответы на эти вопросы!

Ключевые выводы

- Поддержка веб-доступности важна для некоторых, но полезна для всех.
- Сообщество людей с ограниченными возможностями составляет третью по величине экономику в мире с точки зрения покупательной способности, что делает доступность сильным экономическим аргументом.
- Многие правительства закрепляют гарантии доступности в законодательстве, поэтому их соблюдение является также юридическим требованием.

- Организация W3C WAI разработала ряд рекомендаций по обеспечению доступности веб-контента, которым должны следовать команды разработчиков ПО. Приступая к разработке проекта, проверьте самую последнюю версию руководящих принципов.
- Экосистема доступности является всеобъемлющей и включает в себя инструменты и технологии за пределами Интернета. Попробуйте хотя бы одну вспомогательную технологию, например программу чтения с экрана, — этот опыт поможет вам придумать, как улучшить поддержку доступности в своем приложении.
- Реализация поддержки доступности должна интегрироваться в жизненный цикл разработки с самых начальных этапов, потому что реализация в конце цикла станет сущим кошмаром.
- Многие платформы веб-разработки имеют встроенные инструменты для поддержки доступности.
- Тестирование доступности можно реализовать на ранних этапах разработки ПО с помощью чек-листов, автоматизированных инструментов статического тестирования и тестирования во время выполнения, таких как Codelyzer, Pa11y CI, Lighthouse, WAVE и axe-core.
- Вы можете обратиться в организации, предоставляющие услуги по ручному тестированию с помощью пользователей, в том числе с ограниченными возможностями, чтобы получить отзывы о своем приложении.

Тестирование межфункциональных требований

Понимающие межфункциональные
требования по-настоящему понимают
качество!

Компании часто думают о сотнях функциональных требований, стремясь повысить ценность своих продуктов для клиентов и получить доход. Эти функциональные требования составляют основу предлагаемых клиентам бизнес-услуг, таких как возможность забронировать билет с помощью приложения бронирования или совершить платеж посредством приложения интернет-банкинга. Однако простой реализации таких функциональных требований недостаточно для успеха. Представьте, что вы хотите заказать такси и вам нужно подождать 5 мин, чтобы увидеть список доступных вариантов. За это время вы вполне могли бы поймать такси самостоятельно, так зачем же использовать приложение? Или, может быть, приложение функционально, хорошо справляется со своей задачей, но, чтобы заказать такси, требует выполнить несколько шагов. Сложность разочаровывает, и вы, скорее всего, начнете искать более удобную альтернативу. Аналогично если вы узнаете, что приложение раскрывает ваши личные данные, то наверняка избавитесь от него. Это лишь несколько примеров того, почему предприятиям и командам разработчиков ПО необходимо уделять внимание межфункциональным требованиям (Cross-Functional Requirement, CFR). Они делают приложение завершенным и, самое главное, высококачественным.

CFR — это функции приложения, которые должны быть встроены в каждую предлагаемую им возможность. Вот, например, пара CFR для приложения заказа такси: приложение должно давать ответ в течение x секунд, пользователи должны иметь возможность выполнять любые действия не более чем за n шагов, а приложение должно безопасно передавать и хранить данные пользователей. Только когда CFR будут созданы и тщательно протестированы по всем функциям, приложение получит шанс стать сильным конкурентом на рынке.

МЕЖФУНКЦИОНАЛЬНЫЕ НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Часто можно услышать, как межфункциональные требования (CFR) называют *нефункциональными требованиями* (Non-Functional Requirements, NFR). Я, как и многие другие в индустрии программного обеспечения, предпочитаю термин «*межфункциональные*» (<https://oreil.ly/bvITy>), потому что он подчеркивает, что требования распространяются на все приложение и должны реализовываться и тестироваться как часть каждой пользовательской истории и функции. Кроме того, название «нефункциональные требования» может создать ложное впечатление, что эти требования необязательные, а это полностью противоречит цели, к которой мы стремимся, создавая высококачественное приложение!

Если вы привыкли считать требования к программному обеспечению функциональными или нефункциональными, то увидите, что некоторые «функциональные» особенности, такие как аутентификация и авторизация, в этой главе называются межфункциональными. Это связано с тем, что они охватывают все приложение. Например, мы должны проверять подлинность каждого запроса на обслуживание и отвечать только соответствующей информацией с учетом уровня доступа/разрешениями запрашивающего.

Мы уже обсуждали CFR в предыдущих главах, рассматривая такие темы, как тестирование производительности, безопасности, доступности, визуальное тестирование и тестирование данных, а данная глава призвана привлечь внимание ко всему спектру этих требований. Мы рассмотрим CFR в более широком смысле и обсудим общую стратегию тестирования CFR, способную обеспечить непрерывную обратную связь. Обсудим также некоторые важные методологии и инструменты тестирования, которые помогут в реализации стратегии тестирования.

Введение

В главе 1 мы поговорили о том, как представляют себе качество программного обеспечения бизнес и заказчики, и отметили, что обе стороны определяют длинный список атрибутов качества. Эти атрибуты, по сути, превращаются в стандартные межфункциональные требования для любого приложения. В табл. 10.1 перечислены 30 распространенных межфункциональных требований с определениями и примерами.

Таблица 10.1. Список межфункциональных требований с простыми определениями

CFR	Простое определение
Доступность для людей с ограниченными возможностями	Способность системы обеспечивать беспрепятственный доступ к приложению пользователям с ограниченными возможностями, как обсуждалось в главе 9, например, посредством поддержки программ чтения с экрана
Архивируемость	Способность системы хранить историю событий и транзакций, например историю онлайн-заказов пользователя, и по мере необходимости извлекать ее

CFR	Простое определение
Возможность аудита	Способность системы фиксировать бизнес-события и состояния приложения с помощью логов, записей в базе данных и т. д. Как объяснялось в главе 7, эта функция помогает защититься от угрозы отказа
Аутентификация	Способность системы допускать к сервисам приложения на всех уровнях только аутентифицированных пользователей, например, с помощью простой функции входа
Авторизация	Способность системы ограничивать доступ к сервисам приложения на основе разрешений, например допускать к просмотру реквизитов счета только определенных сотрудников банка
Доступность для обращения к системе	Способность системы обслуживать клиентов в течение определенного интервала времени, как обсуждалось в главе 8
Совместимость	Способность двух или более систем действовать в тандеме, не нарушая работу друг друга. Например, способность приложения работать с более ранней версией того же сервиса (так называемая обратная совместимость)
Соответствие	Соответствие системы законодательным требованиям и отраслевым стандартам, таким как WCAG 2.0
Конфигурируемость	Способность системы изменять поведение приложения с помощью переменных, например возможность настраивать тип многофакторной аутентификации
Согласованность	Способность системы давать согласованные результаты в распределенных средах без потери информации, например способность отображать комментарии к сообщениям в социальных сетях в правильном порядке независимо от географического местоположения конечного пользователя
Расширяемость	Способность системы подключать новые функции, например добавлять в приложение новые методы оплаты
Встраиваемость	Способность системы устанавливаться на поддерживаемые платформы, такие как операционные системы и браузеры
Функциональная совместимость	Способность системы взаимодействовать с приложениями, работающими на разных технологиях и платформах. Например, способность системы управления сотрудниками интегрироваться с системами страхования, управления заработной платой, оценки эффективности и т. д.
Локализация/интернационализация	Возможность распространения приложения в разных регионах с предоставлением удобного пользовательского интерфейса на языке конечного пользователя. Например, amazon.de локализован для немецкоязычных пользователей. Это межфункциональное требование также часто называют l10n/i18n по тем же причинам, что и a11y (см. примечание на стр. 316)

Таблица 10.1 (продолжение)

CFR	Простое определение
Удобство сопровождения	Простота поддержки приложения в долгосрочной перспективе благодаря читаемому коду, тестам и т. д. Примером может служить выбор осмысленных имен методов
Мониторинг	Способность системы собирать данные о своей деятельности и предупреждать об ошибках или о том, что показатели выходят за допустимые пределы. Например, рассылка оповещений, когда сервер останавливается
Наблюдаемость	Способность системы анализировать информацию, собранную системами мониторинга, с целью отладки приложения и получения представления о его поведении, например, чтобы понять, как используется каждая функция в пиковые часы, дни, недели и т. д.
Производительность	Способность системы вовремя реагировать на запросы пользователя даже в периоды пиковой нагрузки. Например, приложение заказа такси должно дать ответ пользователю через <i>x</i> секунд даже при пиковой нагрузке
Переносимость	Возможность доставки приложения в новые среды, например интеграция с новой базой данных или облачной средой
Конфиденциальность	Способность системы защищать личные и конфиденциальные данные пользователя, например шифровать данные кредитной карты перед сохранением их в базе данных
Восстановимость	Способность системы восстанавливаться после сбоев, например, за счет автоматизированного резервного копирования данных
Надежность	Способность системы прощать ошибки и непрерывно поддерживать работоспособность сервисов и точность данных. Так, многие приложения включают механизмы поддержки повторных попыток для обработки сбоев в сети и в других подсистемах
Отчетность	Способность системы представлять содержательные отчеты бизнесу и конечным пользователям на основе накопленных событий. Например, Amazon позволяет пользователям создавать отчеты по истории заказов
Устойчивость	Способность системы обрабатывать ошибки и простои. Например, решения по балансировке нагрузки могут пересылать запросы только серверам, находящимся в сети
Повторное использование	Способность системы повторно применять код приложения и сервисы для реализации новых функций. Например, компоненты пользовательского интерфейса могут повторно задействоваться в нескольких пакетах корпоративных приложений
Масштабируемость	Способность системы справляться с распространением на новые регионы, увеличением количества пользователей и т. д. Например, большинство облачных провайдеров предусматривают возможность автоматизированного масштабирования, которая гарантирует добавление дополнительных вычислительных ресурсов при увеличении нагрузки

CFR	Простое определение
Безопасность	Способность системы устранять уязвимости и защищаться от потенциальных атак с помощью инструментов и методов, обсуждаемых в главе 7
Поддерживаемость	Способность системы поддерживать новых разработчиков, присоединяющихся к командам, и новых пользователей, подключающихся к приложению. Примером может служить автоматизация этапов настройки базы кода и набора тестов
Тестируемость	Способность системы моделировать тестовые сценарии и экспериментировать с приложением. Например, возможность создавать имитации для сторонних сервисов с целью моделирования тестовых сценариев и интеграционного тестирования
Удобство использования	Способность системы предоставлять пользователю интуитивно понятный, содержательный и простой интерфейс, например единообразный макет навигации с панелью заголовка

Это не исчерпывающий список, могут быть и другие. В совокупности межфункциональные требования определяют *эксплуатационные* и *эволюционные* качества приложения. Под эксплуатационными понимаются качества, связанные с поведением приложения во время выполнения, такие как готовность к обслуживанию, аутентификация, мониторинг и т. д. Под эволюционными качествами понимаются удобство сопровождения, масштабируемость, расширяемость и т. д., то есть все, что относится к качеству статического кода приложения. Когда приложение не отличается высокими эксплуатационными качествами, конечные пользователи и бизнес сразу ощутят это. Когда приложение страдает недостатками в эволюционных качествах, первыми это ощутят команды разработчиков, а вслед за ними и бизнес. Например, конечные пользователи расстраиваются, когда система не готова к обслуживанию, а когда код трудно поддерживать, недовольны члены команды, что приводит к снижению их продуктивности. Чтобы избежать подобных разочарований, команды должны установить набор межфункциональных требований к приложению в самом начале разработки и постоянно тестировать их на протяжении всего цикла поставки, подобно функциональным требованиям.

Чтобы помочь в этом, мы сейчас обсудим общую стратегию тестирования CFR.

Стратегия тестирования CFR

Для начала обсудим модель FURPS, используемую для классификации всех требований к программному обеспечению¹. Мы будем применять ее для определения высокоуровневой стратегии тестирования межфункциональных требований.

¹ Разработана в Hewlett-Packard и описана Робертом Грейди в книге *Practical Software Metrics for Project Management and Process Improvement* (Prentice-Hall).

Аббревиатура FURPS расшифровывается как *functionality, usability, reliability, performance и supportability* (функциональность, удобство использования, надежность, производительность и поддерживаемость). Эти компоненты можно описать следующим образом.

Функциональность

К этой категории требований относятся пользовательские сценарии в приложении, такие как путь входа, путь проверки возможности сделать заказ и путь его оформления.

Удобство использования

В эту категорию входит набор требований, влияющих на взаимодействие с пользователем, таких как качество изображения, совместимость с браузером, доступность для людей с ограниченными возможностями, простота применения и т. д.

Надежность

Эти требования обеспечивают согласованность, отказоустойчивость и возможность восстановления после сбоев.

Производительность

Эти требования относятся к ключевым показателям эффективности бэкенда и показателям производительности фронтенда, как описано в главе 8.

Поддерживаемость

В эту категорию входят все эволюционные качества кода, такие как удобство сопровождения, тестируемость, безопасность и т. д.

Таким же образом можно представить межфункциональные требования, перечисленные в табл. 10.1. Например, доступность для людей с ограниченными возможностями, как обсуждалось в главе 9, проявляется через функциональные возможности, такие как добавление субтитров в видео, а также через дизайн приложения. Как результат, подход к тестированию доступности будет включать методы и инструменты, используемые для тестирования функциональности и удобства применения. Аналогично одним из способов обеспечения безопасности является добавление функций аутентификации, таких как вход в систему, и реализация мер защиты в статическом коде.

В этом разделе представлены стратегии тестирования для каждой из этих пяти составляющих (рис. 10.1). Чтобы сформулировать стратегию тестирования межфункциональных требований для конкретного проекта, проанализируйте различные их аспекты и примените методы и инструменты, соответствующие приоритетам вашего проекта.

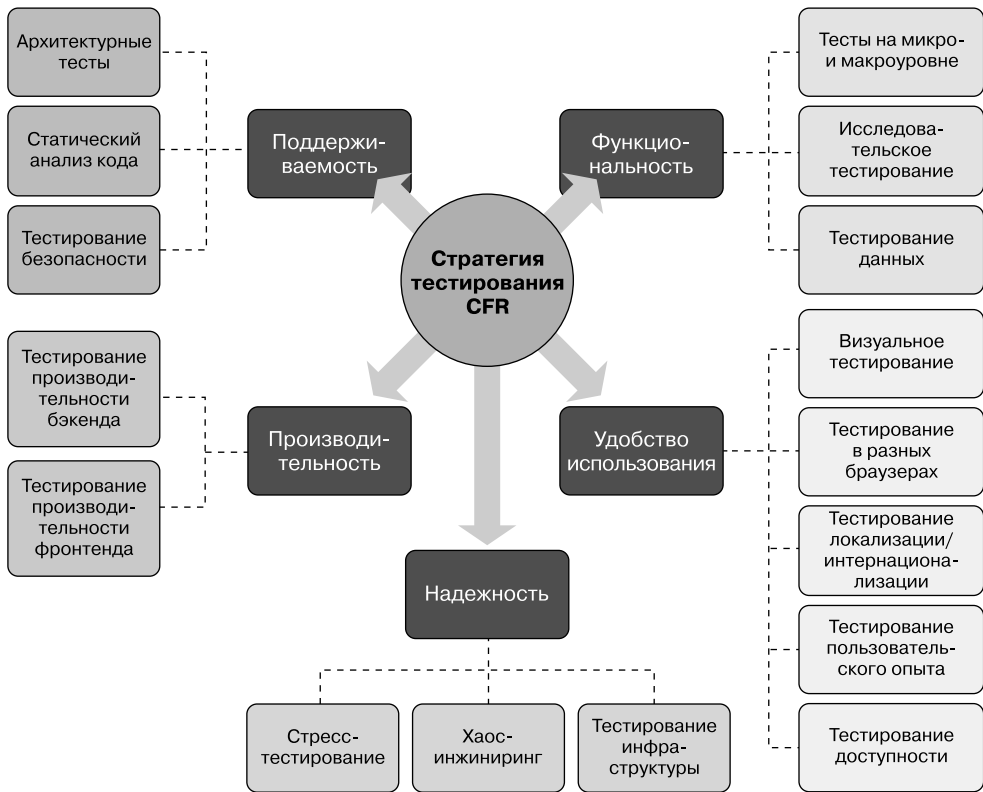


Рис. 10.1. Тестирование пяти составляющих межфункциональных требований

Функциональность

Для проверки функциональных аспектов CFR на разных уровнях приложения можно использовать инструменты и методы ручного исследовательского и автоматизированного функционального тестирования, обсуждаемые в главах 2 и 3. Как вы наверняка помните, для автоматизации тестирования и получения постоянной обратной связи в этом случае можно задействовать такие инструменты, как Postman, Selenium WebDriver, REST Assured и JUnit, а также инструменты и методы тестирования данных, обсуждаемые в главе 5.

Особое внимание при тестировании функциональности, связанной с такими требованиями, как строгая аутентификация клиентов, которая является частью PSD2 или GDPR, следует уделить сбору правильной информации об этих правилах и привлечению команды юристов. Для быстрого ознакомления с нормативными требованиями в следующий раздел главы включен посвященный им подраздел «Тестирование на соответствие».

Удобство использования

Для методичного тестирования удобства применения можно разложить понятие удобства на несколько аспектов, таких как визуальное качество, совместимость с браузерами, локализация/интернационализация, организация взаимодействия с пользователем и доступность для людей с ограниченными возможностями. Инструменты и подходы к тестированию визуального качества и совместимости с браузерами мы обсудили в главе 6, а тестирование доступности — в главе 9, поэтому здесь рассмотрим тестирование остальных аспектов.

Тестирование локализации/интернационализации

Тестирование локализации можно выполнить несколькими способами. Если пользовательский интерфейс для разных языков различается, то следует выполнить визуальное тестирование. Если интерфейс не меняется, а меняются только язык и форматы отображения дат, времени, денежных единиц и т. п., то можно положиться на модульное и ручное тестирование. Например, кроме проверки формата дат и денежных единиц, можно добавить модульные тесты, сравнивающие файлы с переводами строк и проверяющие наличие всех соответствующих ключей. Однако при изменении языка иногда меняется длина текста, что может привести к изменениям в макете пользовательского интерфейса. В таких случаях можно снова применить методы визуального тестирования.

Ручное тестирование текста на конкретном языке следует выполнять в определенной последовательности, чтобы избежать дублирования усилий. Первый шаг — получить осмысленный текст для всех элементов, сообщений и так далее от человека, знающего язык, а затем одобрение от владельца продукта или любого утвержденного бизнесом представителя. Далее необходимо задокументировать правильный текст в каждой пользовательской истории, чтобы обеспечить возможность ее разработки и ручного тестирования. Когда эти шаги пропускают, разработчикам ничего не остается, кроме как перевести текст с помощью систем онлайн-перевода, что приводит к ненужной трате сил и времени на тестирование до и после получения утвержденных строк.

Важно помнить: откладывая тестирование локализации до момента выпуска, вы рискуете получить неправильный макет пользовательского интерфейса на поздних этапах цикла поставки, так как есть вероятность того, что переведенный текст не будет вписываться в макеты элементов, как уже упоминалось.



Функциональные тесты на основе пользовательского интерфейса не следует применять для проверки всего текста в приложении, потому что это делает их очень медленными. Используйте эти тесты только для проверки функциональных потоков в разных локалях, если они различаются. В таких случаях можно повторно задействовать функциональные тесты на основе пользовательского интерфейса, параметризовав строки, используемые в проверяемых утверждениях, и идентификаторы элементов, при условии, что общая стратегия соответствует пирамиде тестирования.

Пользовательский опыт

К пользовательскому опыту (User eXperience, UX) относятся все аспекты приложения, связанные с дизайном, например: насколько интуитивно понятен пользовательский интерфейс, сколько раз нужно щелкнуть кнопкой мыши, чтобы получить необходимую информацию, передают ли значки правильное значение, по вкусу ли цветовая палитра конечному пользователю и т. д. Такие аспекты исследуются в начале проекта и учитываются в дизайне. Например, работая над мобильным приложением интернет-магазина, мы обнаружили, что итальянцы предпочитают яркие цвета, например ярко-красный, и разработали приложение в такой цветовой палитре.

Как правило, проверку аспектов, связанных с восприятием пользователей, следует включать в ручное исследовательское тестирование для каждой пользовательской истории. Группа Нильсона и Нормана провела обширное исследование пользовательского опыта и составила список из десяти эвристик, характеризующих удобство применения (<https://oreil.ly/pPfpY>), проверку которых можно включить в тестирование. Чаще всего в нем участвуют владельцы продуктов и UX-дизайнеры. Для тестирования пользовательского опыта существуют свои инструменты, такие как UserZoom и Optimal Workshop, которые можно использовать для тестирования прототипов дизайна с реальными конечными пользователями. Я видела, как такие тесты, проводимые периодически в течение цикла поставки с различными группами конечных пользователей, приводили к значительному улучшению дизайна.

А/В-тестирование — еще один способ получения информации о восприятии продукта пользователями в режиме реального времени. Правильнее было бы назвать этот процесс не тестированием, а экспериментированием: он включает в себя представление разных вариантов пользовательского интерфейса двум группам конечных пользователей и сбор информации об их поведении, опираясь на которую команда разработчиков сможет принять решение об окончательном дизайне. Например, можно провести простой эксперимент, чтобы понять, на какой кнопке «Продажа» с большей вероятностью щелкнет клиент — красной или синей. В таком эксперименте красные и синие кнопки предъявляют разным группам пользователей в промышленной среде, а данные об использовании собирают и анализируют в течение заданного периода. Для такого рода экспериментов могут потребоваться дополнительные средства и знания в области науки о данных (<https://oreil.ly/DhF5k>), и обычно над ними работает команда, состоящая из владельцев продукта, специалистов по данным, разработчиков и дизайнеров пользовательского интерфейса.

Надежность

Как показано в табл. 10.1, к межфункциональным требованиям, способствующим обеспечению надежности приложения, относятся восстановимость, устойчивость, контролируемость, архивировемость, отчетность, мониторинг, наблюдаемость

и согласованность. Многие аспекты надежности, такие как обработка ошибок, механизмы повторных попыток, резервные механизмы для единственных точек отказа, меры по обеспечению согласованности данных и интеграция со сторонними инструментами для мониторинга и наблюдения за сервисами отчетности, могут восприниматься как пользовательские сценарии. Для их тестирования можно задействовать приемы функционального тестирования, описанные в главах 2 и 3. Также для тестирования надежности можно использовать следующие методы.

Хаос-инжиниринг

Хаос-инжиниринг (chaos engineering) помогает выявить недостатки приложения, способные привести к сбоям, отключениям и другим аварийным ситуациям в нем. Обычно этот метод раскрывает неизвестные прежде недостатки и чрезвычайно полезен в крупномасштабных системах. Хаос-инжиниринг подробно обсуждается в следующем разделе главы.

Тестирование инфраструктуры

Инфраструктура — одна из многих важных частей приложения, помогающих обеспечить надежность и восстановление после сбоев. Если произойдет отказ инфраструктуры, то откажет все. Кроме того, уровень инфраструктуры должен быть настроен таким образом, чтобы поддерживать возможности автоматического масштабирования, оповещения/мониторинга, балансировки нагрузки и архивирования. Целенаправленное тестирование инфраструктуры еще не получило широкого распространения, однако оно начинает набирать обороты из-за растущей потребности бизнеса в масштабировании. Эту тему мы тоже обсудим в следующем разделе главы.

Производительность

Мы обсудили важность производительности и выбор инструментов и показателей для тестирования производительности бэкенда и фронтенда в главе 8. Поэтому просто повторим, что к ключевым показателям относятся готовность к обслуживанию, время отклика и параллелизм, а в числе инструментов, которые могут помочь в тестировании производительности, назовем JMeter, WebPageTest и Lighthouse. Также следует отметить, что тестирование производительности перекликается с тестированием масштабируемости, поскольку определяет порог отказа системы, чем помогает повысить надежность приложения.

Поддерживаемость

Под поддерживаемостью подразумеваются все эволюционные качества кода, такие как совместимость, конфигурируемость, расширяемость, встраиваемость, совместимость, переносимость, удобство сопровождения, возможность повторного использования, безопасность и тестируемость. Некоторые из этих межфункцио-

нальных требований, такие как конфигурируемость функциональных возможностей, совместимость с требуемыми протоколами, возможность установки в соответствующих операционных системах, функциональная совместимость и т. д., можно протестировать с помощью методов и приемов функционального тестирования, обсуждавшихся ранее, при правильной настройке среды. Вот некоторые другие подходы к проверке поддерживаемости.

Архитектурные тесты

Архитектурные тесты добавляются для подтверждения архитектурных характеристик, например размещения классов в правильных пакетах (что обеспечивает возможность повторного использования). Эти автоматизированные тесты дают команде обратную связь в случае отклонений от основных архитектурных характеристик, которые были разработаны с учетом требований CFR, таких как повторное применение, переносимость, удобство сопровождения и т. д. Мы обсудим некоторые инструменты, которые можно использовать для написания таких тестов, в подразделе «Тестирование архитектуры».

Статический анализ кода

Многие инструменты выполняют статический анализ кода и дают полезную обратную связь, помогающую повысить удобство сопровождения. Например, Checkstyle (<https://oreil.ly/2OTi1>) проверяет единство стиля оформления программного кода в команде. PMD (<https://pmd.github.io>) сообщает о таких проблемах, как наличие неиспользуемых переменных, пустые блоки `catch`, дублирующийся код и т. д. Он также позволяет команде добавлять свои правила, соответствующие стандартам проекта. ESLint (<https://eslint.org>) — аналогичный инструмент для проверки кода JavaScript на наличие ошибок в коде, а SonarQube (<https://www.sonarqube.org>) — широко распространенный инструмент, помогающий оценить охват кода тестами и сканирующий его в поисках возможных уязвимостей. В предыдущих главах мы обсуждали и другие инструменты статического анализа кода, которые проверяют его безопасность и доступность.

Используя эти инструменты и методы, можно внедрить раннее тестирование межфункциональных требований. Как обсуждалось в главе 4, тестировать эти требования можно постоянно в рамках функционального тестирования в CI, что позволит получать непрерывную обратную связь по всем аспектам качества и тем самым непрерывно поставлять клиентам высококачественное программное обеспечение!

Другие методы тестирования CFR

В этом разделе мы подробнее обсудим некоторые методы тестирования CFR, перечисленные в предыдущем разделе, такие как хаос-инжиниринг, архитектурное тестирование и тестирование инфраструктуры, чтобы помочь вам применить раннее тестирование CFR и обеспечить возможность непрерывной доставки. В конце

раздела будет представлен набор часто применяемых нормативных требований, что поможет вам провести тестирование на соответствие.



Название этого раздела подчеркивает тот факт, что мы уже обсуждали различные методы и инструменты тестирования CFR в главах 5–9.

Хаос-инжиниринг

Надежность приложений — одно из важнейших межфункциональных требований, потому что любой сбой в обслуживании приводит к убыткам для бизнеса. По оценкам исследования (<https://oreil.ly/TIYbl>), проведенного Gartner в 2014 году, стоимость простоя для некоторых организаций колеблется от 140 тыс. до 540 тыс. долларов в час, и я не удивлюсь, если в 2022-м стоимость простоя окажется еще выше. Признавая важность надежности, признанные на рынке продукты, такие как веб-сервисы Amazon, стремятся достичь времени безотказной работы 99,999 %, то есть того, что суммарное время простоя составит всего 5 мин 15 с в год.

В числе факторов, которые могут привести к простоям, можно назвать ошибки в приложении, единственные точки сбоя в архитектуре, проблемы с сетью, сбои оборудования, неожиданно высокие нагрузки и проблемы со сторонними сервисами, от которых зависит приложение. Большинство этих факторов учитывают при проектировании архитектуры, и команды принимают соответствующие профилактические меры во время разработки. Например, для обработки простоев сервиса широко применяется метод экспоненциальной задержки. Согласно этому методу в случае недоступности сервиса частоту запросов к нему уменьшают экспоненциально, чтобы дать время для восстановления. Аналогично во избежание простоев во время обновлений системы часто реализуется сине-зеленая модель развертывания, когда имеется два идентичных рабочих экземпляра, один из которых обслуживает рабочий трафик, а другой обновляется, после чего трафик переключается на обновленный экземпляр. Помимо этих методов, команды используют и другие меры предупреждения простоев, например реплики для распределения высокой нагрузки, инфраструктуру с поддержкой автоматизированного масштабирования, обработку ошибок во входных данных и т. д. Тем не менее, несмотря на все эти усилия, крупномасштабные распределенные системы создают отдельные проблемы с надежностью приложения, выражающиеся в запутанности рабочих процессов, многоуровневых зависимостях, сбоях сторонних сервисов, ошибках в нижестоящих системах и т. д. Их нелегко предвидеть, и они могут привести к простоям.

Рассмотрим гипотетический пример. Команда из 50 человек работала над крупномасштабным распределенным приложением и установила два отдельных экземпляра для обслуживания клиентов в США и Великобритании. Они настроили каждый экземпляр так, чтобы при выходе из строя одного из них трафик перенаправлялся на другой экземпляр, и предусмотрели в приложении возможность об-

работки запросов из обоих регионов. Команда протестировала функциональность и поддержку перенаправления, а также проверила работоспособность приложения под нагрузкой. Однако, когда экземпляр в Великобритании вышел из строя из-за технических проблем и все запросы оттуда были перенаправлены на экземпляр в США, там происходила пиковая распродажа, в результате чего приложение не справилось с трафиком и выдало ошибки всем пользователям. Первопричина была обнаружена позже в одной из сторонних нижестоящих систем, установившей ограничение на количество запросов в час, которая начинала выдавать ошибки при превышении этого ограничения. С практической точки зрения это один из тех крайних случаев, которые трудно предусмотреть. Команда провела всеобъемлющую проверку, но в таких крупномасштабных распределенных системах трудно запомнить все мельчайшие детали!

Эта гипотетическая команда была не единственной, оказавшейся в такой ситуации. В Netflix тоже столкнулись с неприятностями, когда их сервис мигрировал в облако: облачные экземпляры сталкивались с незапланированными простоями из-за различных проблем, что приводило к убыткам и увеличению рабочего времени инженеров. Они восприняли это как вызов и начали намеренно имитировать сбои и решать проблемы в своем приложении одну за другой, пока оно не стало абсолютно устойчивым к незапланированным сбоям. Для достижения этой цели был разработан инструмент под названием *Chaos Monkey*, который каждый день в рабочее время отключал один случайный экземпляр кластера, а инженеры принимали меры для ликвидации проблемы. Такой подход гарантировал устранение всех присущих системе непредсказуемых недостатков, что в итоге сделало ее устойчивой и надежной. Основываясь на этом успехе, в компании развили эту практику, назвав ее *хаос-инжинирингом* (*Chaos Engineering*).

Формальное определение из книги *Chaos Engineering*¹ (<https://oreil.ly/n6Yp2>) Норы Джонс и Кейси Розенталь (O'Reilly) выглядит следующим образом:

«Хаос-инжиниринг — это дисциплина проведения экспериментов в распределенных информационных системах с целью укрепления уверенности в способности системы противостоять турбулентным условиям эксплуатации».

Другими словами, хаос-инжиниринг предполагает проведение экспериментов, моделирование ошибок, сбоев и других неожиданных сценариев, а также наблюдение за поведением приложения. Эта практика все шире распространяется в индустрии программного обеспечения, и многие компании адаптировали ее под свои потребности. Исходя из коллективного опыта, фундаментальные характеристики хаос-инжиниринга можно описать так.

- Речь скорее идет об экспериментировании, а не о тестировании, то есть хаос-инжиниринг предполагает не проверку ожидаемого поведения системы при

¹ Розенталь К., Джонс Н. Хаос-инжиниринг. — М., 2020.

встрече с неизвестными проблемами, а скорее наблюдение за ее поведением в неожиданных ситуациях и получение информации.

- Цель экспериментов — обрести уверенность в надежности и устойчивости системы. Вы можете отказаться от экспериментов, если уверены в ее способности справиться с неизвестной турбулентностью.
- Хаос-инжиниринг особенно полезен при разработке крупномасштабных распределенных систем.
- Хаос-инжиниринг не входит в сферу ответственности какой-то одной конкретной роли, например инженеров DevOps или тестировщиков. Это командная работа, в которой все заинтересованные стороны вместе разрабатывают эксперименты, проводят их и отлаживают поведение.

Возможно, проведя эксперименты с созданием хаоса, команда из 50 человек заметила бы проблему ограничения скорости раньше и избежала бы катастрофического сбоя!

Эксперимент с созданием хаоса

Всем, кто решится на эксперименты с созданием хаоса, команда Netflix рекомендует проводить их непосредственно в рабочей среде, потому что в тестовой смоделировать реальные переменные чрезвычайно сложно. Они также рекомендуют предусмотреть возможность прерывания эксперимента и возврата системы в нормальное состояние. Сейчас имеется несколько инструментов, помогающих в написании сценариев экспериментов с созданием хаоса, таких как Chaos Toolkit и ChaosBlade, поэтому вам не придется вручную снижать и повышать производительность.

Чтобы провести эксперимент с созданием хаоса, вместе с межфункциональной командой разработайте гипотезу, ставящую под сомнение надежность приложения. Затем определите гипотезу устойчивого состояния, представляющую прогнозируемое поведение приложения во время эксперимента. Напишите сценарий эксперимента, используя выбранный инструмент, и запустите его в рабочей среде.

Если инструмент сообщит, что эксперимент не удался (то есть гипотеза устойчивости приложения не подтвердилась), то ваша межфункциональная команда сможет приступить к действиям по выявлению и устранению причин этого.

Чтобы вы могли получить представление о том, как работает один из инструментов для проведения экспериментов с созданием хаоса, в примере 10.1 показана простая конфигурация эксперимента для Chaos Toolkit (<https://chaostoolkit.org>) — инструмента с открытым исходным кодом, написанного на Python. Сценарий эксперимента имитирует техническую проблему (в данном случае удаляет конфигурационный файл в текущем экземпляре приложения) и проверяет работоспособность альтернативного экземпляра.

Пример 10.1. Эксперимент для моделирования технической проблемы и наблюдения за поведением приложения

```
{
  "version": "1.0.0",
  "title": "Application should still be up if there are technical issues",
  "description": "When a particular config file is missing, application should
    still be up from another instance",
  "contributions": {
    "reliability": "high",
    "availability": "high"
  },
  "steady-state-hypothesis": {
    "title": "Application is up and running",
    "probes": [
      {
        "type": "probe",
        "name": "homepage-must-respond-ok",
        "tolerance": 200,
        "provider": {
          "type": "http",
          "timeout": 2,
          "url": "https://www.example.com/"
        }
      }
    ]
  },
  "method": [
    {
      "type": "action",
      "name": "file-be-gone",
      "provider": {
        "type": "python",
        "module": "os",
        "func": "remove",
        "arguments": {
          "path": "/path/config-file"
        }
      },
      "pauses": {
        "after": 1
      }
    }
  ]
}
```

Сценарий начинается с описания цели эксперимента и указания того, что его назначение — обеспечить высокую надежность и доступность. Затем определяются гипотеза устойчивого состояния и метод, вызывающий техническую проблему. Этот метод, используя возможности инструмента, удаляет указанный файл и ждет в течение 1 с, после чего инструменту предлагается проверить гипотезу об

устойчивости состояния, для чего с помощью одного из сэмплеров выполняется обращение к URL приложения и проверяется, получен ли ответ с кодом состояния 200 в течение 2 с. Этот эксперимент можно запустить из командной строки.

Предположим, мы проводим эксперимент и он терпит неудачу. При дальнейшем исследовании мы обнаружили, что приложению требуется не 2 с, а 4 с из-за препятствий при перенаправлении на альтернативный экземпляр. Это будет ценная информация, полученная в результате простого эксперимента с созданием хаоса.

Chaos Toolkit предоставляет множество других функций для проведения различных экспериментов, их легко настроить в виде файлов JSON, подобных показанному в примере 10.1. Он также может генерировать отчеты HTML по окончании экспериментов.

Тестирование архитектуры

В начале любого проекта обдумывается и составляется список функциональных и межфункциональных требований, а также разрабатывается подходящая архитектура. Предположим, что для удобства сопровождения и возможности повторного использования в архитектуре приложения предусмотрено несколько уровней. Кроме того, поскольку производительность приоритетна, на нужных уровнях размещены механизмы кэширования. Но есть универсальный закон, который может разрушить эти хорошо продуманные решения, известный как закон Конвея. В своей статье *How Do Committees Invent?* (<https://oreil.ly/BR0iT>) Мелвин Конвей утверждает, что структура команды и особенно пути общения между людьми неизбежно влияют на конечную архитектуру продукта. Отдельные команды, работающие над малыми частями большой системы, непреднамеренно оптимизируют свои части, не принимая во внимание общие потребности. Например, команда может предпочесть высокую производительность в ущерб возможности повторного использования, что приведет к необходимости доработки позже. В таких случаях и пригодятся архитектурные тесты, если их правильно спроектировать для защиты основных архитектурных характеристик: они обеспечивают обратную связь командам, когда те отклоняются от генеральной линии.

Для создания этих тестов можно использовать такие инструменты, как ArchUnit (<https://github.com/TNG/ArchUnit>) для Java, NetArchTest (<https://github.com/BenMorris/NetArchTest>) для .NET и др. (<https://oreil.ly/KrikZ>). Например, вы можете добавить архитектурные тесты для проверки циклических зависимостей (чтобы обеспечить удобство сопровождения) или независимости пакетов и возможности их повторного применения. Тесты ArchUnit подобны тестам JUnit и могут выполняться как часть конвейера CI. В примере 10.2 показан тест ArchUnit, который проверяет, все ли классы сервиса управления заказами находятся в пакете `oms`, чтобы гарантировать возможность повторного использования. Соответственно, всякий раз, когда возникает необходимость включить класс, выходящий за рамки обязанностей

сервиса управления заказами, команда будет вынуждена обсудить общую картину и решить, куда этот класс добавить.

Пример 10.2. Тест ArchUnit для подтверждения возможности повторного применения

```
@Test
public void order_classes_must_reside_in_oms_package() {

    classes().that().haveNameMatching("*order*").should().resideInAPackage("..oms..")
        .as("order classes should reside in the package '..oms..'")
        .check(classes);
}
```

Аналогично JDepend (<https://oreil.ly/HechHx>) позволяет получить показатели качества архитектуры с точки зрения расширяемости, удобства обслуживания и возможности повторного применения. Этот инструмент выполняет статический анализ кода всех классов Java и дает различные оценки для данного пакета Java. (NDepend (<https://www.ndependent.com>) — аналогичный инструмент для .NET.) JDepend использует количество абстрактных классов и интерфейсов в пакете как меру его расширяемости, проверяет зависимости от внешних пакетов и выдает предупреждения при обнаружении нежелательных и циклических зависимостей. Тесты JDepend можно писать так же, как и тесты JUnit, и интегрировать их в CI для получения непрерывной обратной связи о качестве архитектуры.

В примере 10.3 показан тест JDepend, который проверяет, зависят ли пакеты A и B друг от друга, создавая циклические зависимости и тем самым затрудняя повторное использование.

Пример 10.3. Тест JDepend для проверки циклических зависимостей пакетов

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class PackageDependencyCycleTest extends TestCase {
    private JDepend jdepend;

    protected void setUp() throws IOException {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/A/classes");
        jdepend.addDirectory("/path/to/project/B/classes");
    }

    public void testAllPackages() {
        Collection packages = jdepend.analyze();
        assertEquals("Cycles exist",
            false, jdepend.containsCycles());
    }
}
```

Аналогичным образом можно писать тесты для проверки наличия в пакете только ожидаемых зависимостей или полного их отсутствия, благодаря которым команды смогут постоянно получать уведомления при возникновении нежелательных изменений в критических характеристиках архитектуры.

Тестирование инфраструктуры

Термин *«инфраструктура»* в общем случае относится к вычислительным ресурсам (например, машинам, виртуальным машинам, контейнерам), сетевым структурам (например, VPN, записям DNS, прокси, шлюзам), ресурсам хранения (AWS S3, SQL Server, системы управления конфиденциальными данными) и т. д., необходимым для бесперебойной работы приложения. Тестирование инфраструктуры включает в себя тестирование этих ресурсов и их конфигураций. Это довольно молодая область тестирования.

Потребность в тестировании инфраструктуры в основном обусловлена растущей потребностью в масштабировании приложений, которую испытывают успешные компании, желающие быстро распространить свои онлайн-услуги на новые регионы и начать обслуживать большее количество клиентов. Чтобы обеспечить такое быстрое масштабирование, они должны иметь возможность быстро скопировать существующий технологический стек приложений, включая настройки инфраструктуры, по возможности автоматически, одним щелчком кнопки мыши. У большинства групп разработчиков имеется автоматизированный процесс тестирования, сборки и развертывания приложения в любой среде одним щелчком мыши, но они не всегда обладают одинаковыми возможностями с точки зрения инфраструктуры, что снижает их способность к быстрому масштабированию. Именно здесь практика *«инфраструктура как код»* (Infrastructure as Code, IaC) становится особенно полезной.

Термин *IaC* относится к практике проектирования инфраструктуры и ее конфигурации в виде многократно применяемого кода, подобного прикладному коду, что обеспечивает возможность непрерывной доставки и масштабируемости. Например, можно написать код, использующий API облачного провайдера для развертывания облачного экземпляра с 3 Гбайт памяти и настройки правил балансировки нагрузки и брандмауэра для конкретного приложения. Этот код необходимо протестировать, чтобы его можно было задействовать для развертывания новых экземпляров инфраструктуры при высокой нагрузке или для экспансии в новые регионы.

Terraform (<https://www.terraform.io>), созданный в HashiCorp, — широко известный инструмент с открытым исходным кодом для создания и выполнения сценариев управления инфраструктурой с использованием декларативного стиля программирования. Сценарии Terraform могут работать с несколькими поставщиками облачных услуг. Далее перечислены некоторые моменты, которые следует учитывать

при тестировании кода инфраструктуры, написанного с помощью Terraform, на различных этапах пути к промышленной эксплуатации.

- Terraform предоставляет команду `terraform validate` для проверки синтаксических ошибок в коде сценариев, которую можно применять уже на этапе разработки.
- TFLint (<https://oreil.ly/VffGT>) — плагин для Terraform, который может помочь выполнить статический анализ кода инфраструктуры, выявить устаревший синтаксис и отклонения от рекомендаций, таких как соглашения об именах, и т. д. TFLint также может проверить, предлагаются ли указанные типы образов популярными поставщиками облачных услуг, такими как AWS, Azure и т. д.
- В ходе поэтапной разработки Terraform может сравнивать последние изменения в коде с текущим состоянием среды и выдавать список предполагаемых изменений в инфраструктуре в качестве меры предосторожности перед их выполнением. Например, если изменения в коде приведут к непреднамеренному удалению базы данных, то функция предварительного просмотра может спасти ситуацию! Список возвращает команда `terraform plan`. Также можно писать автоматизированные тесты, которые выполняют эту команду и проверяют определенные аспекты, например соответствие политике безопасности.
- Следующий шаг — развертывание кода инфраструктуры для создания реальных облачных экземпляров и проверки наличия в них предполагаемых ресурсов (например, работает ли экземпляр в частной подсети и имеет ли он необходимое дисковое пространство). Эти тестовые сценарии можно автоматизировать с помощью таких инструментов, как Terratest, AWSSpec, Inspec и Kitchen-Terraform, и добавить в CI.
- Далее следует сквозное тестирование компонентов инфраструктуры. На этом этапе нужно проверить, могут ли компоненты взаимодействовать друг с другом ожидаемым образом, например, может ли веб-сервер вызывать сервисы приложений. Такое тестирование обычно выполняется после успешного развертывания приложения и благополучного завершения всех функциональных тестов. Его можно провести и до развертывания приложения, написав инфраструктурные тесты с использованием комбинации упомянутых ранее инструментов.

Киф Моррис, автор книги *Infrastructure as Code*¹ (<https://oreil.ly/7wGPq>) (O'Reilly), предполагает, что создание тестов инфраструктуры на различных уровнях может привести к образованию ромбовидной структуры вместо пирамиды. Это связано с тем, что модульные тесты для проверки низкоуровневого декларативного кода, например кода Terraform, могут оказаться бесполезными и их количество рекомендуется уменьшить до минимума. То есть тесты инфраструктуры должны добавляться на соответствующие уровни.

¹ Моррис К. Программирование инфраструктуры. — СПб., 2024.

Когда речь идет об инфраструктуре, помимо сквозного функционального тестирования, необходимо также провести тестирование ряда аспектов, перечисленных далее.

Масштабируемость

Обязательно протестировать автоматическое масштабирование экземпляров в зависимости от нагрузки и убедиться, что после масштабирования функции приложения работают без сбоев.

Безопасность

Безопасность инфраструктуры — важнейшая цель тестирования. С помощью таких инструментов, как Snyk IaC (<https://oreil.ly/vHjYX>), можно проверить потенциальные уязвимости в инфраструктуре во время разработки. Некоторые сценарии тестирования безопасности, такие как проверка непреднамеренно открытых портов, соответствующих общедоступных и частных экземпляров и т. д., можно выполнить вручную или написать автоматизированные тесты инфраструктуры.

Соответствие

Иногда код инфраструктуры должен соответствовать определенным нормам и правилам. Например, для совместимости с PCI DSS (описывается в следующем разделе) необходимо настроить соответствующие межсетевые экраны. Для проверки соблюдения правил HashiCorp предлагает инструмент Sentinel (<https://oreil.ly/sbK6J>) корпоративного уровня.

В Terraform имеется и инструмент с открытым исходным кодом для проверки соответствия — `terraform-compliance` (<https://oreil.ly/IazoT>). Он основан на Python и для написания тестов предоставляет поддержку разработки на основе поведения, аналогичную предоставляемой Cucumber. Инструмент применяет тесты к выводу команды `terraform plan`, а не к реальным экземплярам.

Работоспособность

Необходимо протестировать и все другие эксплуатационные функции, такие как архивирование логов для контроля, интеграция с инструментами мониторинга, функции автоматического обслуживания и т. д.

В зависимости от сложности и характера кода инфраструктуры можно написать автоматизированные тесты для всех этих случаев и интегрировать их в CI. Многие инструменты для автоматизированного тестирования инфраструктуры продолжают развиваться и требуют навыков программирования, выходящих за рамки одного языка. Например, Terratest использует GoLang, `terraform-compliance` — Python, а AWSpec — Ruby. Кроме того, многие инструменты автоматизированного тестирования могут потребовать наличия реальной инфраструктуры, что влечет за собой дополнительные затраты. Учитывая эти ограничения, вы можете разработать стратегию тестирования инфраструктуры, соответствующую потребностям вашего приложения.

Тестирование соответствия

Две правовые нормы часто используются в Интернете — GDPR (<https://gdprinfo.eu>) и WCAG 2.0. Стандарт WCAG 2.0 мы подробно обсудили в главе 9, поэтому здесь коснемся только GDPR, а затем кратко рассмотрим некоторые правовые нормы, связанные с платежами, о которых вам следует знать.



В этом разделе дается лишь краткое введение в эти нормативные требования. За более подробной консультацией по вопросам, касающимся программного обеспечения, я рекомендую обратиться к юристам.

Общий регламент защиты данных

Общий регламент защиты данных (General Data Protection Regulation, GDPR) в первую очередь направлен на защиту частных данных граждан ЕС. Если вы хотите продавать товары гражданам ЕС, то ваш веб-сайт должен соответствовать требованиям GDPR. Аналогично, если школа из США принимает на обучение граждан ЕС через свой веб-сайт, она должна соблюдать требования GDPR. Их несоблюдение может повлечь за собой серьезные штрафы, достигающие 4 % годового дохода компании.



В других странах действуют аналогичные законы о защите данных и конфиденциальности. Так, по состоянию на апрель 2022 года в 71 % стран мира (<https://oreil.ly/Vmhgv>) существовало национальное законодательство по защите данных и конфиденциальности. Например, в Канаде действует Закон о защите конфиденциальности потребителей (Consumer Privacy Protection Act, CPPA), а в Великобритании — собственная версия GDPR, принятая после выхода из Евросоюза.

Согласно GDPR, *частные данные* — это любая информация, которая сама по себе или в сочетании с другими сведениями может быть использована для идентификации живого человека. Расовое или этническое происхождение человека, религиозные или философские убеждения, политические взгляды, сексуальная ориентация, генетические данные, биометрические данные, прошлые или настоящие судимости и так далее классифицируются как *конфиденциальные персональные данные*, которые должны тщательно охраняться. Даже многие онлайн-идентификаторы, такие как IP-адреса, MAC-адреса, идентификаторы мобильных устройств, cookie-файлы, идентификаторы учетных записей пользователей и другие данные, сгенерированные системой, которые могут применяться для идентификации живого человека, попадают под защиту GDPR. Регламент GDPR рекомендует командам разработчиков для защиты данных придерживаться принципов конфиденциальности по умолчанию (privacy by design) — концепции, разработанной доктором Энн Кавукян (<https://oreil.ly/g6Z4K>), в которой излагаются семь основополагающих принципов, направленных на предотвращение любых событий, нарушающих конфиденциальность.

В числе технических мер, которые вы можете реализовать, назовем защиту хранящихся данных с использованием динамической соли и методов хеширования, шифрование передаваемых данных, соблюдение принципа наименьших привилегий, псевдонимизацию, анонимизацию данных и другие общие меры защиты данных, обсуждавшиеся в главе 7.

GDPR также защищает право пользователей контролировать свои данные разными способами (<https://gdpr.eu/checklist>), например такими.

Право на информирование

Вы должны сообщать пользователям приложения, как будут применяться их персональные данные. Обычно это право регулируется политикой конфиденциальности сайта.

Право на доступ

Пользователи имеют право запросить сохраненные ими личные записи.

Право на удаление информации

Пользователи могут потребовать от владельцев сайта удалить их персональные данные, если нет веской причины для их дальнейшего хранения и обработки.

Право на ограничение обработки

Пользователи могут запретить обработку своих персональных данных. Веб-сайт по-прежнему может хранить данные, но не сможет их обрабатывать.

Право на уточнение данных

Пользователи могут исправлять неполную или неточную информацию на сайте.

Право на переносимость

Пользователи могут получать и повторно использовать свои персональные данные.

Право на возражение

Пользователи могут возражать против задействования их личной информации в целях маркетинга, исследований и статистики.

Права, связанные с автоматическим принятием решений

У пользователей необходимо запрашивать согласие на использование их информации для автоматического принятия решений, например для создания личностного профиля.

Большинство этих требований можно протестировать с помощью приемов функционального тестирования. Например, можно добавить автоматизированные тесты на микро- и макроуровне, чтобы убедиться в отсутствии неявного согласия,

то есть того, что личные данные сохраняются только после получения согласия пользователя, и проверить невозможность попадания личной информации в логи приложения. Для этого прекрасно подойдут приемы и методы тестирования безопасности, обсуждавшиеся в главе 7.

PCI DSS and PSD2

Если ваше приложение предполагает прием платежей кредитными картами (что характерно для большинства интернет-магазинов) или предоставляет платежные услуги в регионе ЕС, то оно должно соответствовать двум нормативным документам.

Стандарт безопасности данных в индустрии платежных карт (PCI DSS)

PCI DSS — это глобальный стандарт, установленный Советом по стандартам безопасности платежных карт (<https://oreil.ly/43oIW>) для защиты онлайн-транзакций по картам. Этот стандарт применим к любой организации, которая хранит, обрабатывает или передает данные держателей карт. То есть этот стандарт применим ко всем сайтам, принимают данные кредитных карт, в том числе даже к сайтам, принимающим благотворительные пожертвования. PCI DSS — это не юридическое требование, а обязательный стандарт, которому должны соответствовать банки и торговые точки, осуществляющие транзакции по картам. За его несоблюдение предусмотрены штрафы согласно соответствующим контрактам между компаниями и платежными системами. Обычно компании могут подтвердить свое соответствие с помощью анкеты для самооценки.

PCI DSS включает 12 рекомендаций (<https://oreil.ly/yOOwE>) по обеспечению безопасности транзакций в приложении, таких как шифрование передаваемых данных, наличие брандмауэра, обновление антивирусного программного обеспечения и т. д. Соответственно, приступая к тестированию, вам следует продумать сценарии защиты данных платежных карт, такие как маскирование данных в пользовательском интерфейсе и во всех местах хранения, ограничение доступа к данным, отказ от хранения данных в логах и т. д. Здесь пригодится упражнение по моделированию угроз, о котором рассказывалось в главе 7.

Директива о платежных услугах (PSD2)

PSD была первой реализованной директивой о платежных услугах (<https://oreil.ly/cu4gd>) в регионе ЕС, главной целью которой было предотвращение преступлений, связанных с онлайн-платежами. Другая ее цель — усилить конкуренцию в индустрии платежей, чтобы не допустить монополизации платежных услуг банками. PSD2 — это обновленная версия исходного стандарта PSD. Соблюдение требований предусмотрено законодательством ЕС для всех поставщиков платежных услуг. Если вы создаете приложение, предоставляющее платежные услуги клиентам в ЕС, то вам следует обратить внимание на правила PSD2.

PSD2 в основном фокусируется на функциях строгой аутентификации клиентов (Strong Customer Authentication, SCA) (<https://oreil.ly/QLPX1>) и на расширении

охвата PSD2 внутри и за пределами ЕС. Например, директива оговаривает обязательное соответствие ее требованиям, если хотя бы один этап транзакции включает в себя государство — член ЕС. Чтобы соответствовать требованиям PSD2, предприятия могут выбрать поставщика платежных услуг, такого как Stripe (<https://oreil.ly/sW1VL>) или PayPal (<https://oreil.ly/iGule>), или встроить функции SCA в свои приложения. Проще говоря, SCA можно приравнять к многофакторной аутентификации. Европейская комиссия определяет SCA как механизм аутентификации, который проверяет как минимум два из следующих трех элементов:

- какие-либо уникальные сведения о пользователе, например пароль;
- уникальные данные о чем-то, чем владеет пользователь, например о дебетовой или кредитной карте либо мобильном устройстве;
- уникальные биометрические характеристики пользователя, такие как изображение лица, звукозапись голоса или отпечаток пальца.

Такие функции необходимо тщательно протестировать, чтобы гарантировать их соответствие требованиям PSD2.

Таким образом, первый шаг в проверке соответствия — детальное изучение законодательства. Затем необходимо выбрать подходящие приемы и методы тестирования CFR, обсуждаемые в разделе с описанием стратегии надлежащего фулстек-тестирования (где охватываются пять аспектов). После того как приложение будет протестировано и готово, команда юристов или уполномоченный орган должен приступить к сертификации соответствия. И только при успешной сертификации цикл испытаний на соответствие можно считать завершенным.

Благодаря такому подходу вы сможете протестировать длинный список межфункциональных требований, предъявляемых к вашему приложению, обеспечить успешное его развитие и дать своей команде возможность осуществлять непрерывную доставку, сместив тестирование CFR на ранние этапы разработки ПО.

Перспективы — эволюционное развитие и испытание временем

Мы обсудили, как повысить качество приложения путем тестирования функциональных и межфункциональных требований. Однако важно понимать, что требования к программному обеспечению не закрепляются навечно в начале проекта. Как было упомянуто ранее, требования к программному обеспечению постоянно меняются вместе с потребностями рынка, и это неизбежно. Также нужно принять как факт то, что новые требования почти всегда ставят под угрозу существующую реализацию, если ими не руководствоваться разумно. Например, член команды может поспешно отменить шифрование, чтобы повысить производительность и тем самым создать серьезную угрозу безопасности приложения.

Это основная предпосылка книги Нила Форда, Ребекки Парсонс и Патрика Куа *Building Evolutionary Architectures*¹ (O'Reilly) (<https://oreil.ly/yIaBp>), в которой авторы определяют новое межфункциональное требование: *эволюционность* — способность системы сохранять существующие архитектурные характеристики (например, многоуровневую организацию, методы сохранения данных, шифрование при хранении и передаче), которые облегчают выполнение заданного набора функциональных и межфункциональных требований, одновременно включая новые изменения. Для поддержки эволюционного развития они рекомендуют предусмотреть подходящие защитные условия для основных архитектурных характеристик, которые не могут нарушаться, что обеспечит командам мгновенную обратную связь при любом отклонении от них. Эти защитные условия могут иметь форму автоматизированных тестов по каждому функциональному и межфункциональному требованию (например, тесты производительности, сканирование безопасности, результаты аудита доступности для лиц с ограничениями, архитектурные тесты и функциональные тесты на микро- и макроуровне), а также показатели охвата кода тестированием, статическим анализом и т. д. Этот ансамбль тестов и показателей, которые вместе называются *функциями приспособленности* (fitness functions), поможет командам вносить поэтапные изменения без ущерба для существующей реализации и создавать эволюционирующую архитектуру.

В заключение отмечу, что все взгляды, методы и инструменты функционального и межфункционального тестирования, которые мы изучали на протяжении всей книги, включая и эту главу, в совокупности помогают построить эволюционную архитектуру, способную выдержать испытание временем, в дополнение к получению высокого качества уже сегодня!

Ключевые выводы

- Межфункциональные требования, обычно называемые нефункциональными требованиями, так же важны для успеха приложения, как и функциональные. Функциональные и межфункциональные требования в совокупности превращают приложение в высококачественный продукт.
- Межфункциональные требования в основном определяют эксплуатационные и эволюционные качества приложения.
- Межфункциональные требования применимы к широкому спектру функций приложения и поэтому должны разрабатываться и тестироваться как часть каждой пользовательской истории. Наличие чек-листа CFR в каждой пользовательской истории может помочь успешно протестировать CFR.

¹ Форд Н., Парсонс Р., Куа П. Эволюционная архитектура — СПб.: Питер, 2018.

- Модель FURPS абстрагирует все требования к программному обеспечению в форме пяти компонентов. Нетрудно заметить, что CFR полностью соответствуют этим компонентам.
- В главе представлена стратегия тестирования для каждого из пяти компонентов модели FURPS, которую можно использовать для определения общей стратегии тестирования CFR в конкретном проекте. Эта стратегия должна уделять внимание каждому CFR индивидуально, исходя из потребностей проекта.
- Сместите тестирование CFR на ранние этапы разработки ПО, автоматизировав эти тесты и интегрировав их в CI.
- Хаос-инжиниринг — это метод проведения экспериментов, позволяющий выявить недостатки приложения, которые могут сделать его ненадежным. Эксперименты следует проводить итеративно всей командой.
- Такие инструменты, как ArchUnit и JDepend, помогают подтвердить архитектурные характеристики приложения, чтобы сохранить некоторые эволюционные качества кода.
- Тестирование инфраструктуры — это новая область тестирования. Оно необходимо в случаях, когда есть потребность в быстром масштабировании приложения. Инструменты автоматизированного тестирования инфраструктуры все еще развиваются, и их применение может повлечь за собой дополнительные затраты с точки зрения расширения знаний и фактического развертывания тестов инфраструктуры.
- GDPR и WCAG 2.0 — это правила, часто применяемые в веб-приложениях. Чтобы проверить соответствие им, может потребоваться их доскональное знание, для чего необходимо консультироваться с командой юристов.
- Функциональные и межфункциональные тесты становятся функциями приспособленности (fitness functions) и не только помогают командам создавать высококачественное программное обеспечение сегодня, но и конструировать эволюционные архитектуры, способные выдержать испытание временем.

Тестирование мобильных приложений

Представьте себе день без мобильного телефона!

С момента появления смартфонов мобильные устройства стали для многих из нас дополнительной конечностью. Они привнесли новые возможности в нашу жизнь, доставляя повседневные услуги к нашему порогу в результате одного прикосновения и движения. Я не могу вспомнить ни одного иного объекта, способного выполнять те же функции, что и смартфон. Мы используем их для покупки продуктов, одежды, бытовой техники и других предметов первой необходимости. С их помощью читаем книги, смотрим фильмы и играем в игры. Смартфоны облегчают банковские операции, оплату счетов и организацию наших дел. Более того, они дают нам чувство психологической безопасности, потому что мы знаем: чтобы получить помощь, достаточно сделать один звонок!

Учитывая все эти преимущества и возможности применения, неудивительно, что во всем мире насчитывается 6,6 млрд пользователей смартфонов (<https://oreil.ly/HvCHF>). Но удивительно то, что в мире более 8 млрд мобильных абонентов (<https://oreil.ly/IEGtR>) — значительно больше, чем людей на планете! Масштабы распространения ошеломляют, но исследования соотносят эти цифры с широким использованием. Например, одно недавнее исследование показало, что американцы заглядывают в свои телефоны в среднем 344 раза в день (<https://oreil.ly/td87T>), или каждые 4 мин. Средний владелец смартфона пользуется 10 приложениями в день и 30 приложениями в месяц (<https://oreil.ly/fffg3>). И такое широкое применение смартфонов не ограничивается какой-то конкретной возрастной группой: по оценкам, молодежь от 18 до 24 лет тратит на смартфоны 93,5 ч каждый месяц, люди от 45 до 54 лет — 62,7 ч, старше 65 лет — 42,1 ч (примерно 3, 2 и 1,5 ч в день соответственно). Таково влияние смартфонов на всех нас.

При таком широком использовании неудивительно, что, по состоянию на 2021 год в ведущих магазинах приложений Google Play и Apple App Store было доступно 5,7 млн приложений (<https://oreil.ly/6IMp3>), и в ближайшие годы их количество будет лишь расти, потому что эти приложения приносят прибыль бизнесу. Только в 2020 году доходы от мобильных приложений во всем мире составили

более 318 млрд долларов (<https://oreil.ly/zbvwc>), а, по прогнозам, к 2025 году эта цифра превысит 613 млрд долларов.

Почему эти цифры важны? Потому что мы, разработчики и тестировщики ПО, собираемся разрабатывать и тестировать мобильные приложения и это явный призыв к оттачиванию наших навыков. Цель главы — дать представление о приемах и инструментах тестирования мобильных приложений. Если вам интересно узнать, чем тестирование мобильных приложений отличается от тестирования веб-приложений, то здесь вы получите ответ на этот вопрос. Эта глава знакомит с общей мобильной средой и особенностями тестирования мобильных и веб-приложений. Вашему вниманию будет представлена стратегия тестирования мобильного уровня, включая автоматизированное функциональное тестирование, тестирование производительности, безопасности, доступности, визуальное тестирование и тестирование CFR. Кроме того, в этой главе вы найдете упражнения, которые помогут быстро освоиться и подготовиться к работе над мобильными проектами!

Введение

Для начала взглянем на общую мобильную среду, характерные проблемы и особенности, требующие нашего внимания при тестировании мобильных приложений.

Введение в мобильный ландшафт

Как показано на рис. 11.1, в мобильной среде следует учитывать три основные области: устройства, сами приложения и сеть. Рассмотрим их по очереди.

Устройства

В ходе эволюции мобильные устройства стали отличаться друг от друга по нескольким параметрам. Выбирая устройства для тестирования, очень важно понимать, каковы эти аспекты. Как правило, следует стремиться охватить тестированием не менее 85 % целевых устройств. Вот список различных характеристик устройства, которые следует учитывать при выборе стратегии тестирования.

Размер экрана

К мобильным устройствам относятся планшеты и смартфоны. В мире более миллиарда человек используют планшеты (<https://oreil.ly/37EwY>), и это слишком большое число, чтобы пройти мимо него! Учитывая разнообразие формфакторов и моделей устройств, неудивительно, что размеры экранов планшетов (<https://screensiz.es/tablet>) и смартфонов (<https://screensiz.es/phone>) существенно различаются. Более того, размер экрана одного и того же устройства меняется со сменой его ориентации (то есть в альбомном или книжном режиме просмотра), а современные телефоны позволяют просматривать несколько приложений в режиме разделения экрана, дополнительно разделяя доступное экранное пространство.

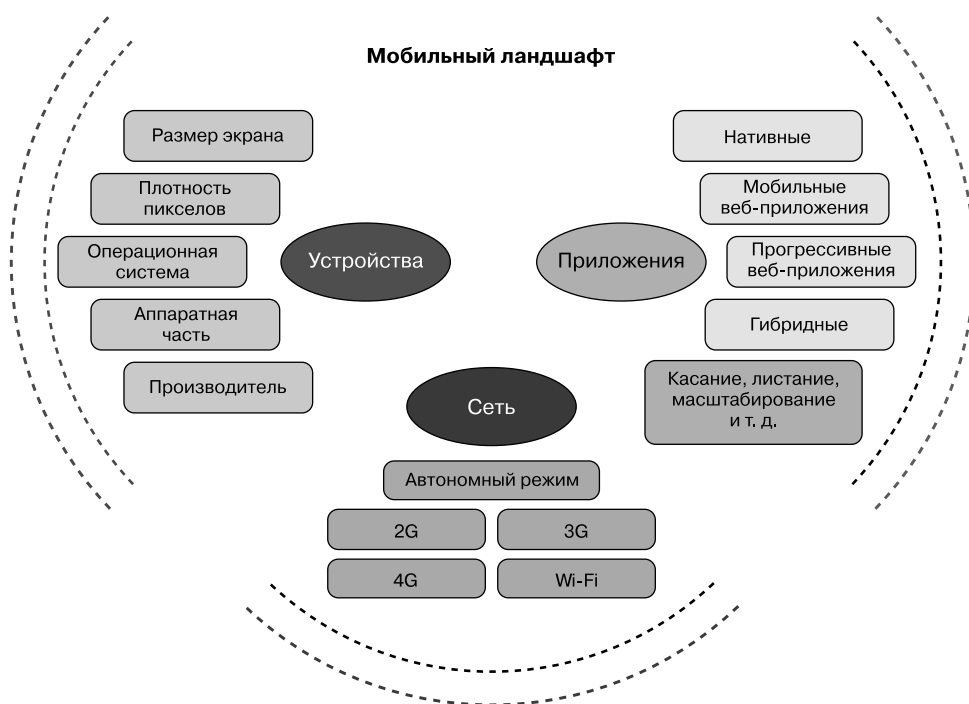


Рис. 11.1. Мобильный ландшафт

Размер экрана оказывает большое влияние на общее впечатление конечного пользователя, поэтому проектирование, разработка и тестирование приложений на устройствах с разными размерами экранов имеют большое значение в мобильной среде. Например, на экранах маленького размера пользователю может потребоваться воспользоваться прокруткой, чтобы увидеть всю страницу, тогда как на экранах большого размера может оставаться много пустого пространства. И то и другое способно вызывать неудовольствие пользователя.

Плотность пикселей

Пиксел — это небольшая квадратная область на экране, несущая часть отображаемой информации, а плотность пикселей — это количество пикселей, помещающихся на одном квадратном дюйме экрана. Чем больше плотность, тем выше качество изображения. Мобильные устройства различаются не только размерами экрана — при одинаковых размерах экрана они могут иметь разную плотность пикселей (<https://oreil.ly/eXIz7>). В зависимости от плотности пикселей есть устройства с низкой, средней, высокой, сверхвысокой, сверхсверхвысокой и сверхсверхсверхвысокой плотностью¹. Эта характеристика мобильных

¹ Также можно встретить обозначения LDPI, MDPI, HDPI, XHDPI, XXHDPI и XXXHDPI, где DPI расшифровывается как dots per inch («точек на дюйм»).

устройств особенно сильно влияет на качество изображений, так как на некоторых из них размер изображений автоматически изменяется в соответствии с изменением размера экрана, что приводит к размытию или искажению. Следовательно, изображения должны создаваться с учетом плотности пикселей, и этот аспект необходимо включить в тестирование.



Разрешение экрана — это показатель количества пикселей, которые он может отображать по горизонтали и вертикали. Например, экран с разрешением 1024×768 в альбомной ориентации может отображать 1024 пиксела по горизонтали и 768 по вертикали.

Операционная система

Точно так же, как в мире настольных компьютеров есть Windows, macOS, Linux и другие операционные системы, в мире мобильных устройств есть Android, iOS, Windows Mobile, Symbian, KaiOS и т. д. Как вы наверняка знаете, лидирующие позиции занимают Android и iOS. На их долю приходится ~99 % мобильных устройств (<https://oreil.ly/ZWMvn>) по всему миру. Но это еще не все! Существует множество версий каждой ОС, которые до сих пор официально поддерживаются и работают. Это явление называется *фрагментацией*. Например, по состоянию на 2020 год Android 6.0 (<https://oreil.ly/OnKm8>) по-прежнему оставалась второй по широте применения версией Android, несмотря на то что она вышла в 2015-м, а первое место занимала версия Android 9.0. Таким образом, в тестирование должны вовлекаться разные версии ОС, потому что некоторые из них могут не поддерживать определенные функции или обрабатывать их по-разному.

Аппаратное обеспечение

Аппаратные конфигурации мобильных устройств — ОЗУ, ЦП, емкость аккумулятора, емкость локальной памяти и т. д. — это еще один параметр, который меняется от модели к модели. Аппаратное обеспечение влияет на производительность приложения с точки зрения параллельной обработки, скорости отображения и общего удобства использования. Когда приложение зависит от возможностей устройства, таких как поддержка GPS, наличие камеры, микрофона, сенсорного экрана и других аппаратных датчиков, впечатления конечных пользователей будут различаться.

Этот параметр важен, потому что он может повлиять на основные функции приложения. Например, мобильное приложение, предназначенное для сбора информации о выживших во время стихийных бедствий, таких как цунами или циклоны, не может полагаться на наличие у волонтеров высококачественной камеры и должно стараться не потреблять слишком много энергии аккумулятора. В зависимости от сферы предполагаемого использования может потребоваться спроектировать, разработать и протестировать приложение с учетом возможностей аппаратного обеспечения.

Производитель устройства

В настоящее время на рынке присутствуют несколько производителей устройств: Oppo, Samsung, Xiaomi, LG, Motorola, Google, Apple и т. д. Некоторые из них имеют свои версии Android, например Cyanogen OS, Oxygen OS и Hydrogen OS. Кроме того, каждый производитель устройства придерживается собственной конфигурации оборудования, например предоставляет центральную аппаратную кнопку «Домой» или кнопку «Назад». Эти нюансы необходимо учитывать при разработке и тестировании мобильных приложений.

Очевидно, что соображения, связанные с самими устройствами, создают массу проблем для команд разработчиков ПО. А теперь посмотрим на проблему через призму приложения.

Приложение

Ключевая особенность мобильных приложений — разнообразный набор поддерживаемых ими способов взаимодействия. В дополнение к стандартному щелчку и вводу текста, которые поддерживаются веб-приложениями, в мобильном приложении можно задействовать такие жесты, как листание, касание, длительное касание, изменение масштаба, нажатие и перетаскивание, вращение и многие другие! Эти жесты и взаимодействия во многом делают использование мобильных устройств более привлекательным и персонализированным. Подмножество взаимодействий можно сделать общедоступным в приложении, например, по жесту листания слева направо на любой странице можно отображать меню, а по жесту листания снизу вверх от нижней части экрана — вызывать список дополнительных возможностей. Такие общие взаимодействия являются межфункциональными требованиями для всего приложения, и их необходимо проектировать, реализовывать и тестировать как часть каждой пользовательской истории. Однако возможности расширенного взаимодействия могут ограничиваться в зависимости от типа приложения. Мы, конечные пользователи, возможно, никогда не задумывались об этих различиях между мобильными приложениями, но нам, как разработчикам ПО, необходимо знать об этом, чтобы соответствующим образом подходить к тестированию. В настоящее время широко используются следующие четыре типа мобильных приложений.

Нативные

Нативные (native) приложения разрабатываются в основном для работы на одной мобильной ОС, например Android или iOS. Преимущества выбора разработки нативных приложений заключаются в их способности обеспечить максимальную производительность, доступ к аппаратному обеспечению устройства, ко всем функциям и API ОС (включая жесты), работу в автономном режиме и согласованный и гармоничный внешний вид. Нативные приложения для Android обычно пишутся на Java или Kotlin, а для iOS — на Objective C

или Swift. Они распространяются через Google Play и Apple App Store соответственно. Каждая из этих платформ распространения определяет свои правила соответствия и процедуры утверждения после выпуска приложения, поэтому может возникнуть задержка, прежде чем приложение станет доступным для общественности. Обычно это не проблема, однако даже срочные исправления ошибок проходят процедуру утверждения, что задерживает их выпуск! Еще одним существенным недостатком является стоимость разработки, поскольку для каждой целевой операционной системы необходимо разрабатывать отдельное нативное приложение.

Мобильные веб-приложения

Мобильные веб-приложения — это веб-сайты, доступ к которым получают с помощью мобильных веб-браузеров. Их преимущества заключаются в том, что они не зависят от ОС, не требуют установки и места на локальном диске для установки. Они также не требуют одобрения со стороны магазинов приложений. Более того, эти приложения можно разрабатывать с использованием обычных технологий веб-разработки, таких как HTML5 и CSS, поэтому нет необходимости изучать языки, специфичные для мобильных ОС. Но у них есть свои недостатки: отсутствие доступа к таким функциям ОС, как телефонная книга, камера и т. д., и они не могут работать в автономном режиме. В результате пользовательский опыт очень ограничен.

Гибридные

Гибридные приложения сочетают в себе лучшие черты нативных и веб-приложений. Гибридное приложение разрабатывается с использованием стандартных веб-технологий, таких как HTML, JavaScript или CSS, а затем помещается в отдельный контейнер, обеспечивающий доступ к API операционной системы. В числе популярных фреймворков для разработки гибридных приложений можно назвать React Native, Ionic, Apache Cordova и Flutter. Самое интересное, что они позволяют запускать одно и то же приложение в нескольких ОС. Гибридные приложения необходимо отправлять в магазин приложений для распространения, но веб-элементы можно размещать на сервере и получать по сети. В результате легко обновлять эти части приложения в обход процедуры утверждения в магазине. Но это ограничивает возможность использовать гибридные приложения в автономном режиме, так что команды обычно хранят минимальный набор контента локально на устройстве, чтобы дать возможность работать в автономном режиме. В целом гибридный подход упрощает разработку и снижает затраты на нее. Однако за это приходится платить производительностью, которая у гибридных приложений ниже, чем у нативных. Кроме того, поскольку эти приложения обычно создаются для работы в разных ОС, может возникнуть непреднамеренный побочный эффект в виде отчуждения некоторых конечных пользователей, привыкших работать со своими приложениями определенным образом в предпочитаемой ими ОС.

Прогрессивные веб-приложения

Прогрессивные веб-приложения (Progressive Web Apps, PWA) — это расширенные версии мобильных веб-приложений. Они доступны для установки по известному URL и занимают очень мало места в локальном хранилище. Несмотря на то что это все-таки веб-приложения, они могут поддерживать push-уведомления, работать в автономном режиме и использовать функции ОС, что делает их похожими на нативные приложения. С точки зрения производительности PWA находятся на одном уровне с нативными приложениями, а так как это все же веб-приложения, они могут работать в разных ОС и браузерах. Более того, реализация их возможностей требует меньших затрат по сравнению с нативными и гибридными приложениями. Учитывая эти преимущества, сейчас PWA стали предпочтительным выбором для бизнеса. В 2017 году Twitter заменил свое мобильное веб-приложение прогрессивным веб-приложением (<https://oreil.ly/ukF4b>), в результате чего показатель отказов уменьшился на 20 %, количество отправленных твитов увеличилось на 75 %, а количество просмотренных страниц выросло на 65 % за сессию!

Как вы уже наверняка поняли, выбор типа создаваемого приложения будет определять объем тестирования в таких областях, как поведение в онлайн- и офлайн-режимах, поддержка функций ОС, поведение при обновлении приложения, взаимодействия и т. д. А теперь перейдем к последней важной области — сети.

Сеть

Не все люди на нашей планете имеют доступ к высокоскоростным сетям. Низкая пропускная способность часто становится проблемой в удаленных регионах, но даже в крупных городах подключение к сети не всегда бывает стабильным. Итак, когда приложение зависит от подключения к сети, то, помимо Wi-Fi, вы должны подумать о поддержке разных типов мобильных сетей, например 2G, 3G и 4G, и даже о полной автономности. Вам придется протестировать поведение приложения в таких сценариях, как тайм-ауты, вывод сообщений об ошибках при колебаниях сети (например, при переключении между 4G и 3G), возможность автономной работы, скорость запуска с разными типами сетей и т. д. Возможно, вам даже придется с самого начала разрабатывать приложение с учетом таких ограничений. Например, Facebook выпустил приложение Facebook Lite (<https://www.facebook.com/lite>) главным образом для тех, кто испытывает проблемы с пропускной способностью сети. Оно может работать с сетями 2G и обеспечивает бесперебойную работу при нестабильных сетевых соединениях.

Взгляд на мобильный ландшафт через эти три призмы должен был дать вам некоторое представление о дополнительных сложностях, с которыми приходится иметь дело при тестировании мобильных приложений. А теперь, чтобы лучше понять масштабы их тестирования, углубимся в архитектуру мобильного приложения.

Архитектура мобильного приложения

В главе 2 мы обсуждали архитектуру типичного веб-приложения. Как вы помните, оно имеет веб-интерфейс, который принимает запросы пользователей, и сервисы, обрабатывающие запросы, взаимодействуя при этом с уровнем БД. Архитектура мобильного приложения отличается не сильно. Как показано на рис. 11.2, пользовательский интерфейс мобильного приложения заменяет уровень веб-интерфейса, а все остальное, как правило, почти неизменно.

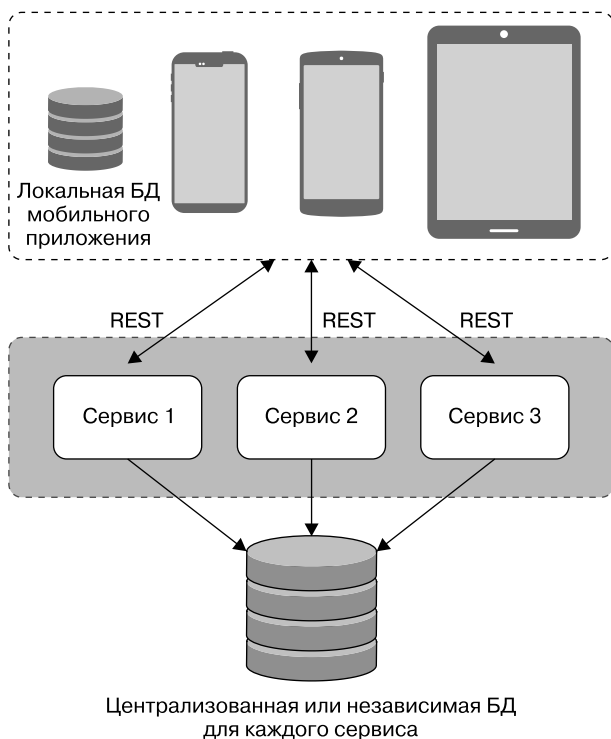


Рис. 11.2. Архитектура мобильного приложения

Обратите внимание на дополнительный компонент — локальную базу данных. Здесь нативные и гибридные приложения хранят отдельные данные, такие как имя пользователя, изображение профиля, последний полученный контент и т. д., для поддержки автономного режима и ускорения отображения интерфейса приложения. В остальном мобильные приложения похожи на веб-приложения: мобильные приложения точно так же вызывают сервисы по сети и принимают запросы от пользователей. Как результат, подходы к тестированию сервисов и уровня базы данных остаются прежними: вы проводите тестирование на микро- и макроуровне и пишете модульные, интеграционные и функциональные тесты,

а также проверяете производительность сервисов, безопасность, соответствие законодательству и другие CFR. Вдобавок к этому вы должны протестировать мобильный пользовательский интерфейс, обращая внимание на присущие ему сложности. Эти конкретные аспекты тестирования мобильного пользовательского интерфейса мы обсудим далее.



Чтобы получить более глубокое представление о внутренних компонентах мобильного пользовательского интерфейса, изучите руководство по архитектуре Android (<https://oreil.ly/evawz>) от Google.

Стратегия тестирования мобильных приложений

Самый первый элемент в стратегии мобильного тестирования, который следует учитывать, — это список устройств, на которых будет производиться тестирование. Цель состоит в том, чтобы обеспечить охват тестированием 85 % целевого сегмента клиентов для каждой пользовательской истории, потому что тестирование на всех вариантах устройств (с разными размерами экрана, ОС, аппаратным обеспечением и т. д.) вряд ли возможно. Например, тестирование на всех версиях Android и устройствах всех производителей окажется слишком трудоемким и дорогостоящим. Особенно отрицательно на темпах разработки скажется необходимость тестирования на десятках устройств для проверки одной пользовательской истории в итеративном процессе разработки, таком как Agile или Scrum. Поэтому сужение списка устройств для тестирования имеет большое значение. Далее приведен набор вопросов, на которые следует найти ответы, чтобы отфильтровать устройства для достижения цели — охвата 85 % (рис. 11.3).

- Какие сегменты клиентов являются целевыми для бизнеса? Например, бизнес по производству высококачественной одежды может быть ориентирован на богатых людей, поэтому из тестирования можно исключить дешевые телефоны.
- На какие конкретные рынки/страны компания пытается выйти и какие ОС и поставщики занимают лидирующие позиции на этих рынках? Например, бизнес по производству одежды может решить сосредоточиться на европейских городах, а в Европе ведущие поставщики — Samsung и Apple (<https://oreil.ly/rfe8S>). Это еще больше сужает список устройств, поскольку высока вероятность того, что состоятельные покупатели будут пользоваться флагманскими устройствами данных производителей.
- Если компания уже присутствует в Интернете, то какая доля обращений к ее приложению приходится на конкретные устройства? Например, может выясниться, что большая часть клиентов пользуется существующим веб-приложением через iPhone и планшеты Samsung.
- Каков диапазон пропускной способности сети, доступной на целевых рынках? Например, средняя скорость мобильной сети в Европе — примерно 54 Мбит/с

(<https://oreil.ly/O2G8e>), что требует включения в тестирование устройств с поддержкой 4G. Критерий пропускной способности сети становится особенно важным при работе с недорогими телефонами.

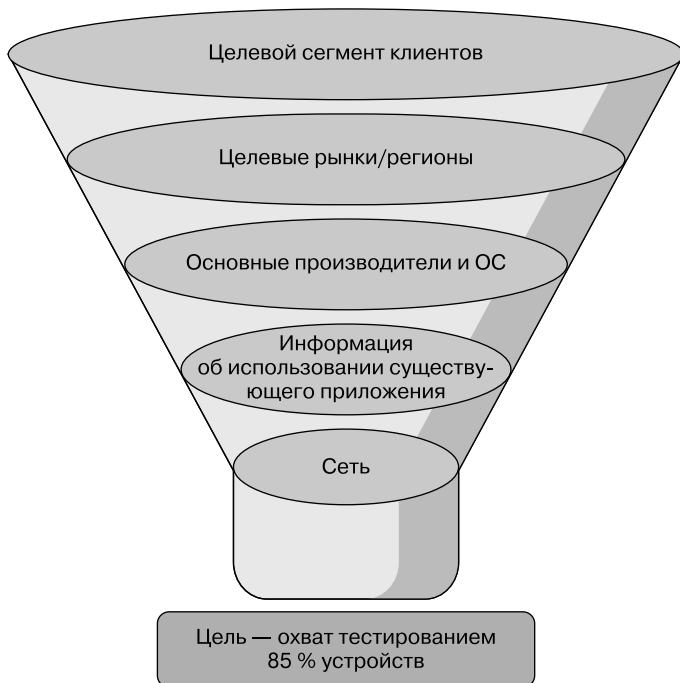


Рис. 11.3. Стратегия отбора мобильных устройств для тестирования

Получив ответы на эти вопросы от компании или представителя продукта, вы сможете выбрать три-четыре телефона с соответствующими характеристиками и, возможно, еще несколько интересных устройств, чтобы проверить их во время регулярных сеансов поиска ошибок.



Выбрать устройства для тестирования нужно в начале проекта. Затем следует проанализировать затраты и решить, стоит их купить или подписаться на облачные услуги таких поставщиков, как AWS Device Farm, Firebase Test Lab, Xamarin Test Cloud, Perfecto или Sauce Labs. Они позволяют запускать автоматизированные тесты на своих реальных устройствах, но имейте в виду, что взаимодействие может оказаться довольно медленным.

На рис. 11.4 показана стратегия тестирования мобильного пользовательского интерфейса. Как видите, методы тестирования аналогичны тем, которые обсуждались на протяжении всей книги до этого момента. Далее мы поговорим о том, как разные методы помогают преодолевать сложности, возникающие в мобильной среде.

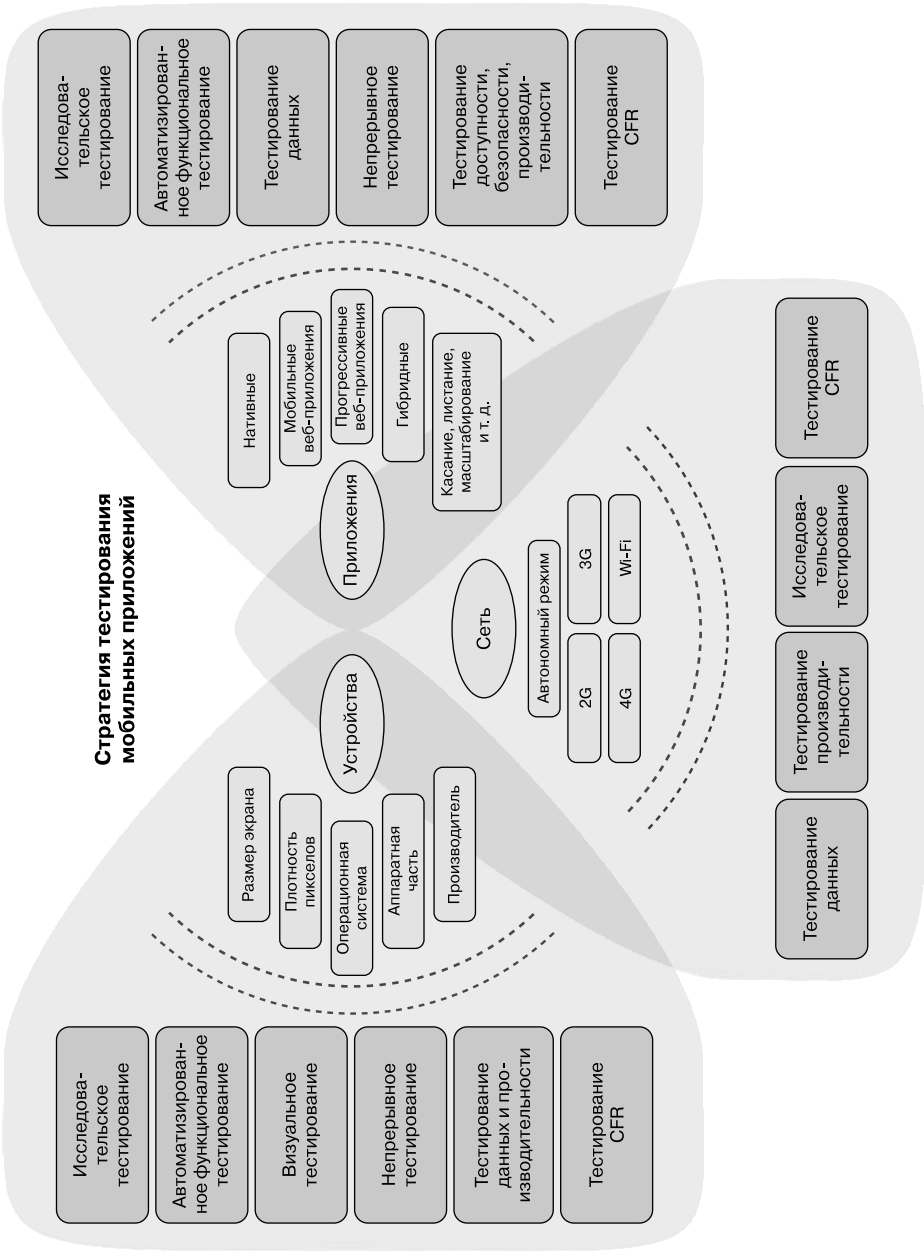


Рис. 11.4. Стратегия тестирования мобильных приложений

Ручное исследовательское тестирование

Исследовательское тестирование становится еще более важным в мобильной среде из-за многообразия комбинаций устройств, приложений и сетей. Методы и стратегия исследовательского тестирования, описанные в главе 2, будут полезны и здесь, и их применение позволит всесторонне проверить поведение приложения. Для облегчения исследования мобильных веб-приложений можно задействовать инструмент Chrome DevTools, позволяющий просматривать веб-сайты в любом разрешении, как было показано в главе 8. Для исследования приложений других типов можно приобрести устройства, которые вы отобрали для тестирования, или применить эмуляторы/симуляторы.

Эмуляторы и симуляторы — это программы, создающие среду, которая имитирует устройство на компьютере. Например, Android Studio предоставляет эмуляторы Android, точно имитирующие аппаратные и программные конфигурации реальных устройств, таких как Google Nexus 4, Samsung Galaxy 5, Moto G и многих других. Аналогично iOS предоставляет симуляторы iPhone и iPad. Далее в этой главе, в разделе с упражнениями, мы рассмотрим настройку эмулятора Android. Эмуляторов и симуляторов может быть достаточно для проверки работоспособности приложения, но не следует забывать, что они не способны имитировать некоторые аппаратные функции, включая определенные сенсорные жесты, интеграцию датчиков и др. Лично я считаю, что тестирования с помощью этих инструментов недостаточно, чтобы признать приложение готовым к выпуску, и отрасль с этим согласна — обе компании, Apple и Google, рекомендуют перед выпуском проводить тестирование на реальных устройствах. Эмуляторы и симуляторы полезны для разработки тестов, так как позволяют выполнять быстрые проверки, но в целом их следует использовать только при отсутствии реальных устройств или для быстрой черновой проверки работоспособности.



Agile-тестирование позволяет легко сместить тестирование устройств на ранние этапы разработки ПО. Например, во время разработки специалист может ориентироваться на самое проблемное разрешение (LDPI или MDPI), а во время тестирования бизнес-аналитик, инженер по качеству и разработчик могут тестировать каждый на своей версии устройства. Для предотвращения регрессии автоматизированные тесты можно запускать на наборе выбранных устройств в CI.

Автоматизированное функциональное тестирование

Тестирование функционального поведения приложения и взаимодействий можно автоматизировать с помощью модульных и сквозных функциональных тестов пользовательского интерфейса. Подобно функциональным тестам для веб-приложений, их необходимо включить в конвейер CI, чтобы получать непрерывную обратную связь. Сквозные тесты можно применять и для проверки работы функций приложения на наборе целевых устройств, запуская их либо в виде дымовых тестов, либо

в виде ночных регрессионных. Appium и Espresso — это два инструмента, широко используемые при разработке сквозных тестов на основе пользовательского интерфейса для Android, а Appium и XCUITest применяются в разработке для iOS. Espresso и XCUITest позволяют тестировать только нативные приложения, тогда как Appium фактически позволяет автоматизировать тестирование приложений всех трех типов (нативных, гибридных и мобильных веб-приложений). Далее в этой главе вы узнаете, как задействовать этот инструмент для создания сквозных тестов на основе пользовательского интерфейса.

Тестирование данных

В мобильных приложениях данные, предназначенные для разных целей, могут храниться на разных уровнях. Как показано на рис. 11.2, существуют локальная база данных мобильного приложения, общая база данных и локальное хранилище. Важно понимать, какие потоки данных связаны со всеми этими хранилищами, и включать их в тестирование, как обсуждалось в главе 5. Например, мобильное приложение социальной сети, такой как Facebook, может хранить последние сообщения в локальной БД мобильного приложения, чтобы в случае нестабильности подключения к сети ее можно было задействовать для быстрого отображения приложения. Рассматривая такие ситуации, вам, возможно, придется подумать о том, как пользователи воспримут отображение устаревшей информации, которая может храниться в локальной БД, как синхронизировать локальную БД и т. д. Кроме того, пользователи могут обращаться к приложению с нескольких устройств, таких как телефон, планшет и веб-браузер на настольном компьютере. Поэтому необходимо протестировать синхронизацию данных с базами данных всех устройств.

В отношении общей БД необходимо отслеживать различные транзакции, совершаемые с разных устройств, и предусмотреть механизм обновления соответствующих данных без конфликтов. Например, пользователь может сохранять события календаря с разных мобильных устройств и синхронизировать их, когда сеть доступна. Необходимо отслеживать эти действия и обновлять информацию в общей базе данных. Проверку таких сценариев тоже можно автоматизировать с помощью функциональных тестов на микро- и макроуровне. Таким образом, двусторонняя синхронизация данных между общей и локальными мобильными базами данных на устройствах, ограниченная условиями сети, может быть важным аспектом для тестирования данных в мобильных приложениях.

Кроме того, если существуют функции, требующие сохранять файлы в локальном хранилище устройства и извлекать их оттуда, их тоже нужно протестировать. Вы должны учитывать граничные условия как для внутреннего, так и для внешнего хранилища устройства (когда хранилище заполнено или недоступно), а также ограничения ОС при обработке файлов различных форматов. В целом, думая о тестировании данных, рекомендуется рассмотреть мобильное приложение со всех трех сторон!

Визуальное тестирование

Визуальное тестирование обычно выполняется при тестировании на устройстве вручную. Как упоминалось ранее, тестирование устройства тоже можно сместить на ранние этапы разработки ПО. А еще вы можете автоматизировать визуальное тестирование для экранов разных размеров (выбранных при выборе устройств) с помощью Appium и Applitools Eyes (<https://oreil.ly/M6pQz>). Applitools Eyes, как обсуждалось в главе 6, — это коммерческий сервис, использующий ИИ для автоматизации визуального тестирования мобильных приложений, а Appium имеет открытый исходный код. Упражнение по автоматизации визуального тестирования с помощью Appium будет включено в эту главу. Вы можете выбирать между двумя инструментами, опираясь на факторы, перечисленные в главе 6.

Тестирование безопасности

В главе 7 мы обсудили образ мышления при тестировании безопасности и основные моменты, связанные с безопасностью, на которые следует обращать внимание при тестировании функциональности. Все, о чем там говорилось, применимо и к тестированию мобильных приложений. Например, во время тестирования следует учитывать шифрование и безопасное хранение конфиденциальных данных пользователей, надежные механизмы аутентификации, разрешения для доступа к другим приложениям на телефоне и т. д.

В главе 7 мы говорили также об инструментах тестирования безопасности, которые автоматически сканируют статический код приложения на наличие уязвимостей и имитируют известные атаки. Как уже упоминалось, эти инструменты применимы и на уровне сервисов, в том числе сервисов мобильных приложений. Возможность автоматизированного статического и динамического сканирования защищенности для проверки мобильного пользовательского интерфейса (Android/iOS/Windows) предоставляется инструментом с открытым исходным кодом под названием Mobile Security Framework (MobSF) (<https://oreil.ly/7MCp9>). GitLab, популярная платформа DevOps, обеспечивает возможность статического тестирования безопасности приложений (SAST) для мобильных приложений (<https://oreil.ly/SUKjm>), реализованную на базе MobSF. Далее в этой главе мы рассмотрим пример использования MobSF и еще одного инструмента автоматизированного сканирования безопасности.

Помимо инструментов автоматизированного сканирования безопасности, команды разработчиков ПО должны знать десять основных рисков мобильных приложений (<https://oreil.ly/zvnFX>), названных организацией Open Web Application Security Project (OWASP), и устранять их во время разработки. В зависимости от уровня квалификации команды может потребоваться привлечь квалифицированных тестировщиков безопасности после разработки.

Наконец, поскольку на момент написания книги тестирование безопасности мобильных приложений было довольно узкой областью, я рекомендую следить за знаниями, накапливаемыми сообществом OWASP в его руководстве *Mobile Security Testing Guide* (<https://oreil.ly/p4903>).

Тестирование производительности

В главе 8 вы узнали, как настроить автоматизированные нагрузочные тесты, стресс-тесты и тесты устойчивости для своих сервисов. Продолжайте их запускать и контролировать.

Производительность на уровне мобильного пользовательского интерфейса может оказаться более важной, чем производительность веб-интерфейса, потому что мобильные приложения обычно работают в среде с не самым производительным процессором, ограниченными объемом памяти и емкостью батареи и неидеальными условиями сети. Для успешного тестирования производительности мобильных устройств обеспечьте два аспекта.

1. Приложение не должно монополизировать или истощать критически важные ресурсы устройства, такие как процессор, память и заряд аккумулятора.
2. Приложение должно быстро реагировать на действия конечного пользователя.

Чтобы протестировать пункт 1, можно задействовать профилировщики для соответствующих ОС — например, Android Profiler (<https://oreil.ly/cHq6p>) в Android Studio и инструменты XCode (<https://oreil.ly/s6GPN>) в iOS. Проверки потребления ресурсов можно добавлять в виде автоматизированных модульных тестов с использованием соответствующих инструментов и интегрировать в конвейеры CI для непрерывного тестирования производительности. Appium тоже предлагает API для получения аналогичных данных о производительности приложений в Android, как будет показано далее в этой главе.

При тестировании пункта 2 обратите особое внимание на такие элементы, как время запуска приложения или время от щелчка на значке приложения до момента его открытия. Оно должно быть меньше 5 с (<https://oreil.ly/cujWm>). Точно так же время реакции на любое действие внутри приложения должно быть менее 3 с, иначе это приведет к увеличению количества повторных попыток выполнить его. Однако на задержки существенно влияет пропускная способность сети, в которой происходят вызовы сервисов. Чтобы измерить время отклика приложения, можно смоделировать различные условия работы сети в эмуляторе или симуляторе. Другая область тестирования производительности — стресс-тестирование. Чтобы нагрузить приложение, можно быстро запустить несколько действий, таких как нажатие нескольких кнопок, увеличение и уменьшение масштаба, отправка запросов, навигация по страницам и т. д., и посмотреть, не произойдет ли сбой. В состав Android входит автоматизированный инструмент под названием Monkey,

помогающий провести автоматизированное стресс-тестирование. Мы рассмотрим его далее в этой главе.

Таким образом, по аналогии со многими другими видами тестирования, размышляя о тестировании производительности мобильных приложений, вы должны рассматривать их со всех трех точек зрения.

Тестирование доступности

W3C WAI предоставляет подробные инструкции по применению WCAG 2.0 к мобильным приложениям (<https://oreil.ly/WpOyC>). Рекомендации по обеспечению доступности мобильных устройств для людей с ограниченными возможностями соответствуют тем же четырем ключевым принципам: приложение должно быть воспринимаемым, работоспособным, понятным и надежным. В число функций, которые необходимо протестировать, входят возможность увеличения и уменьшения масштаба, удобочитаемость на экранах небольшого размера, хороший цветовой контраст между элементами, разумные размеры кнопок, единообразие макета во всем приложении, оптимальное размещение элементов в области просмотра, чтобы не вынуждать пользователя прокручивать экран, и т. д. Обе основные ОС, iOS и Android, предоставляют много инструментов для проверки доступности, которые будут перечислены в следующих пунктах, хотя в целом набор инструментов для автоматизированного тестирования доступности сейчас довольно ограничен.

iOS

iOS предоставляет следующие инструменты для тестирования доступности.

- Средство чтения с экрана VoiceOver доступно как на физических устройствах, так и в симуляторах iOS. Его можно применять для сквозного тестирования пользовательских сценариев.
- В симуляторах iOS имеется инспектор доступности XCode Accessibility Inspector (<https://oreil.ly/lcUcw>) для проверки наличия у элементов соответствующих атрибутов доступности. Его можно использовать для отладки.

Android

Android имеет более развитую поддержку раннего тестирования доступности. Вот некоторые из них, начиная слева:

- Android Studio (<https://oreil.ly/1c2Q9>) — среда разработки, в которой можно настроить вывод предупреждений о различных проблемах доступности во время разработки;

- Espresso (<https://oreil.ly/jFxD>) (и Robolectric до версии 4.5) — инструмент автоматизации тестирования пользовательского интерфейса нативных приложений для Android, позволяющий просканировать каждое представление приложения на предмет поддержки доступности. Эти тесты можно интегрировать с существующим набором тестов Espresso и запускать в CI;
- TalkBack — программа чтения с экрана, встроенная в Android. Ее можно применять для сквозного тестирования пользовательских сценариев;
- сканер доступности Accessibility Scanner (<https://oreil.ly/8cYmG>) — инструмент, проверяющий мобильные приложения на наличие проблем с доступностью. Его можно задействовать на этапе ручного тестирования пользовательской истории.

Кроме того, в Android имеется функция Switch Access, которая позволяет применять внешние вспомогательные устройства для взаимодействия с приложениями (известны как переключатели), а также поддерживает функцию BrailleBack для подключения дисплея Брайля к устройству и голосовой доступ для управления устройством Android с помощью голосовых команд. После передачи приложения в магазин Google Play даже предоставляет командам отчеты о проверке доступности перед выпуском.

Тестирование межфункциональных требований

Межфункциональные требования (CFR), обсуждавшиеся в главе 10, такие как возможность аудита, переносимость, надежность, совместимость и т. д., сохраняют свою актуальность и при тестировании мобильных приложений. Далее перечислены некоторые CFR, помимо безопасности и доступности, на которые следует обратить особое внимание при тестировании пользовательского интерфейса мобильного приложения.

Удобство использования

Если задуматься, мобильное устройство — это очень личный предмет. Как упоминалось ранее, большинство взрослых проводят с телефонами 2–3 ч в день и для многих из нас эти устройства стали дополнительными конечностями. Следовательно, удобство их применения — это серьезная проблема. Громкой рекламой можно убедить конечных пользователей загрузить приложение, но они будут работать с ним постоянно, только если смогут настроить по своему вкусу. Например, конечный пользователь может быть левшой или правой, иметь привычку решать сразу несколько задач в нескольких открытых приложениях, использовать приложения во время вождения, знать несколько языков, предпочитать какой-то определенный тип взаимодействия и т. д. Все эти факторы необходимо учитывать при тестировании удобства применения. Конечно, невозможно удовлетворить все потребности более чем 7,7 млрд человек, тем не менее, тестируя мобильные приложения, вы должны постараться учесть

как можно больше аспектов удобства использования. С этой целью можно реализовать подход к тестированию удобства использования из главы 10. Кроме того, не следует забывать о важности предварительного исследования поведения конечных пользователей на вашем целевом рынке/в регионе. В этом может помочь Google: сайт Think with Google (<https://oreil.ly/me2Vw>) предоставляет подробную инфографику о поведении мобильных пользователей в нескольких странах в дополнение к другим ключевым отчетам, связанным с мобильными устройствами.

Прерывания

Это разновидность межфункционального требования к надежности, характерная для мобильных устройств. Мобильные устройства используются для разных целей, включая обмен сообщениями и телефонные звонки, поэтому работа любого приложения может быть прервана внешними отвлекающими факторами. Типичное поведение пользователя — оставить текущее приложение работать в фоновом режиме, отвлекшись, например, на входящий телефонный звонок или важное уведомление в чате. Покончив с неотложным делом, пользователь возобновляет работу с приложением.

Поэтому при тестировании мобильного приложения важно учитывать возможность прерывания. Что произойдет с текущим запросом, если приложение внезапно окажется в фоновом режиме? Что произойдет с аутентификацией, когда приложение приостановится, а затем возобновит выполнение? Что произойдет с текущим запросом, если приложение будет внезапно завершено или прервано? Что произойдет, если во время работы приложения на устройстве разрядится аккумулятор? Помните, что это — межфункциональное требование и его необходимо протестировать во всем приложении.

Возможность установки и обновления

Установка приложений на различные устройства и ОС из соответствующих магазинов приложений — важный аспект тестирования. Для установки требуется определенный объем локального дискового пространства на устройстве. Кроме того, во время установки приложение может запрашивать у пользователей разрешения на доступ к оборудованию устройства или к другим приложениям (к камере, микрофону, контактам, фотогалерее, сервисам определения местоположения и т. д.). Эти сценарии необходимо протестировать, включая такие случаи, как отсутствие достаточного места в локальном хранилище, отказ пользователя дать разрешения и несовместимость с версией ОС. Кроме того, при тестировании обновления приложения необходимо убедиться, что этот процесс не нарушит работу существующих потоков. Например, изменения в структурах локальной базы данных, если таковые имеются, не должны влиять на существующие функции. Кроме того, пользователь не должен выходить из приложения после обновления. Не забудьте протестировать обновление с более старых версий приложения, а не только с последней. Если при обновлении приложению потребуются новые разрешения, их получение тоже необходимо протестировать.

Поскольку установка и обновление зависят от состояния сети, включите также разные сценарии, связанные с сетью. Точно так же убедитесь в правильной работе функции удаления приложения.

Мониторинг

Сбои в мобильных приложениях, в отличие от веб-приложений, — довольно распространенное явление, поэтому мониторинг в мобильных приложениях является критически важным требованием. Иногда могут возникнуть сложности с воспроизведением условий, вызвавших сбой приложения, и в таких случаях понять проблему могут помочь инструменты мониторинга Firebase Crashlytics, Dynatrace, New Relic и т. д. По этой причине их следует интегрировать в тестовую среду уже на этапе разработки, чтобы упростить отладку сбоев приложения.

Обратите внимание на то, что тестирование некоторых из этих CFR, таких как тестирование установки и обновления на целевых устройствах и ОС или реакции на прерывания (<https://oreil.ly/xIGHC>), можно автоматизировать с помощью функциональных тестов на микро- и макроуровне и получать непрерывную обратную связь — вспомните стратегию непрерывного тестирования из главы 4.

На этом мы завершаем обсуждение стратегии тестирования мобильных приложений. Далее познакомимся с приемами создания наборов тестов с помощью некоторых инструментов, упомянутых в этом разделе.

Упражнения

Приведенные далее упражнения помогут вам настроить фреймворк Java — Appium для создания функциональных и визуальных тестов пользовательского интерфейса. Я выбрала Appium (<https://appium.io>), потому что, как упоминалось ранее, он поддерживает все три типа мобильных приложений (нативные, веб- и гибридные) и несколько ОС.

Appium

Appium — это инструмент с открытым исходным кодом, поддерживаемый активным сообществом. Будучи кросс-платформенным инструментом автоматизации, Appium объединяет фреймворки автоматизации, специфичные для разных ОС, такие как XCUITest (разработан в Apple для iOS) и UiAutomator (разработан в Google для Android), в рамках общего набора оберток API — WebDriver API, с которыми мы познакомились в главе 3. Например, Appium использует класс `DesiredCapabilities` для создания экземпляра драйвера, взаимодействующего с приложением. Кроме того, `findElements(By.id)`, `click()`, `isElementPresent()` и другие API остаются прежними. В результате кривая обучения получается очень пологой для тех, кто имеет опыт автоматизации тестирования с помощью Selenium WebDriver. Кроме того, Appium,

как и WebDriver, не зависит от языка. Это означает, что вы можете писать тесты на любом языке программирования, например Ruby, Python, Java, JavaScript и т. д., используя соответствующие клиентские библиотеки.

В Appium анонсировали выход следующей основной версии 2.0, в которой изменился порядок установки сервера Appium, драйверов автоматизации и плагинов. Например, в Appium 1.x драйверы автоматизации для конкретных ОС включались в состав самого инструмента, тогда как в версии 2.x их необходимо устанавливать отдельно. На момент написания этих строк новая версия находилась на стадии бета-тестирования и должна была выйти в 2022 году. Поскольку это будущее Appium, в данном упражнении мы используем бета-версию 2.x. Мы также станем применять Android, но, поскольку Appium API одинаковы для всех ОС, вы сможете задействовать описанные здесь приемы и для разработки тестов для приложений iOS.

APPIUM — ИНСТРУМЕНТ RPA!

Роботизированная автоматизация процессов (Robotic Process Automation, RPA) (<https://oreil.ly/Qg8jf>) в наши дни является горячей темой в отрасли. Она рассматривается как способ уменьшения нагрузки, связанной с рутинными ручными задачами, и повышения эффективности работы за счет сквозной автоматизации бизнес-процессов. Другими словами, она автоматизирует типичные бизнес-процессы, такие как сбор данных в электронной таблице, ввод их во внутренний инструмент, запуск задания по обработке данных, проверка их появления и т. д.

Appium 1.x в основном используется для автоматизации мобильных приложений, поддерживает автоматизацию настольных приложений для Windows (<https://oreil.ly/XS7PT>) и Mac (<https://oreil.ly/468q8>) с соответствующими драйверами. Кроме того, для взаимодействия с этой версией не требуется, чтобы тестируемое приложение было разработано той же командой, которая создает тесты (то есть ей не нужен исходный код). Таким образом, помимо использования возможностей Selenium WebDriver, можно применять Appium 1.x как инструмент RPA (<https://oreil.ly/yGqUT>)! Надеюсь, эта поддержка сохранится и в Appium 2.x.

Приступим!

Предварительные условия

Предварительные условия аналогичны условиям для других инструментов автоматизации, обсуждавшихся в главе 3, поэтому, прежде чем приступать к установке, проверьте, выполняются ли они. Вам понадобятся:

- Node.js (<https://nodejs.org/en>) для настройки сервера Appium;
- последняя версия Java (<https://oreil.ly/Uq5Wk>) (в этом упражнении мы используем клиентскую библиотеку Java Appium);
- IDE, например, IntelliJ (<https://oreil.ly/y90qz>);
- Maven (<https://oreil.ly/FAOuB>).

Эмулятор Android

После успешной настройки всех предварительных условий создайте эмулятор Android для запуска теста Appium.

1. Загрузите и настройте Android Studio (<https://oreil.ly/5hRn0>). Эта среда разработки включает в себя Android SDK и все необходимые инструменты.
2. В Android Studio выберите пункт меню More Actions ► AVD Manager (Дополнительные действия ► Диспетчер AVD). (AVD означает Android Virtual Device — виртуальное устройство Android.)
3. Нажмите кнопку Create Virtual Device (Создать виртуальное устройство), чтобы просмотреть список существующих профилей оборудования для планшетов, телефонов, устройств Wear OS и т. д. Выберите категорию Phone (Телефон) и затем профиль, например Pixel 2, 5.0 inches (Pixel 2, 5,0 дюйма), нажмите кнопку Next (Далее).
4. Выберите версию Android, например, Android 8.0. Если у вас нет запрошенной версии, будет предложено загрузить ее.
5. На следующем экране укажите имя эмулятора, скажем Oreo, и щелкните на кнопке Finish (Готово). Теперь в списке доступных виртуальных устройств должен появиться эмулятор Android 8.0 Pixel 2.
6. Нажмите кнопку Воспроизвести/Запустить, чтобы запустить эмулятор.

Для этого упражнения вы можете загрузить демонстрационное Android-приложение ApiDemos-debug.apk из репозитория GitHub (<https://oreil.ly/uNfNs>). Установите приложение, перетащив его внутрь эмулятора, затем откройте его и ознакомьтесь с ним.

Установка и настройка Appium 2.0

Чтобы установить и настроить Appium, выполните такие действия.

1. Выполните следующую команду, чтобы установить Appium v2.0:

```
$ npm install -g appium@next
```



Обратите внимание на то, что этот шаг может измениться после выхода официальной версии.

2. Настройте драйвер UiAutomator2, выполнив команду:

```
$ appium driver install uiautomator2
```



Для работы в iOS вам понадобится драйвер XCUITest, который можно установить с помощью команды `appium driver install xcuitest`.

3. Запустите сервер Appium командой:

```
$ appium server -ka 800 -pa /wd/hub
```

4. Загрузите Appium Inspector (<https://oreil.ly/QAXmU>) — инструмент с графическим интерфейсом, который позволяет отыскивать элементы в мобильном приложении.

Рабочий процесс

Как упоминалось ранее, для создания экземпляра соединения с мобильным приложением Appium использует объект `DesiredCapabilities`. В Appium Inspector можно настроить его через графический интерфейс и подключиться к приложению для исследования элементов. Попробуйте исследовать демонстрационное приложение для Android, для чего выполните следующие действия.

1. Откройте инспектор и укажите значения в разделе `Desired Capabilities` (Желаемые возможности) (рис. 11.5). Сохраните настройки, чтобы иметь возможность использовать их позже.

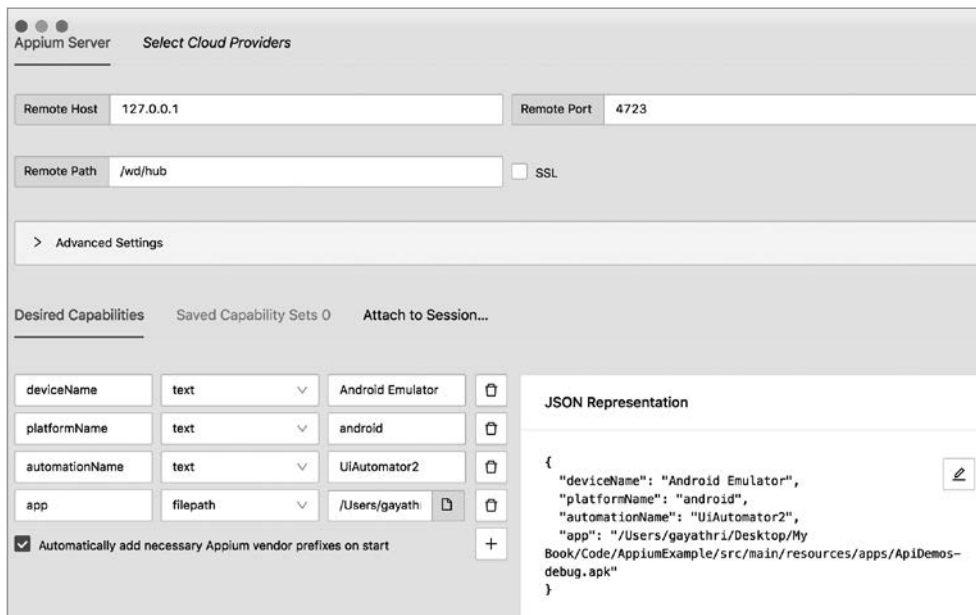


Рис. 11.5. Настройка желаемых возможностей перед подключением к демонстрационному приложению

2. Нажмите кнопку `Start Session` (Начать сеанс). После этого в окне инспектора должно запуститься демонстрационное приложение.

Рассмотрим автоматизацию простого тестового сценария: открыть демонстрационное приложение и убедиться, что второй элемент на домашней странице содержит текст `Accessibility`. Выполните следующие шаги.

1. Откройте IntelliJ и создайте новый проект Maven с именем `AppiumExample`.
2. Добавьте в файл `pom.xml` зависимости `Appium Java` и `TestNG`. (В главе 3 подробно рассказывалось, как это сделать.)
3. Создайте папку `apps` в каталоге `/src/main/resources` и скопируйте в нее файл демонстрационного приложения `ApiDemos-debug.apk`.
4. Создайте пакет `pages` в `/src/main/java`. Создайте пакеты `tests` и `base` в каталоге `/src/test/java`. Пакет `pages` будет содержать классы страниц, а пакет `tests` — классы тестов. В пакет `base` войдут классы настройки.
5. Класс `Base` содержит методы установки и удаления `Appium` с желаемыми возможностями, настроенными с использованием имени пакета приложения, пути к приложению, имени эмулятора, имени устройства, имени платформы и имени фреймворка автоматизации, как показано в примере 11.1.

Пример 11.1. Класс `Base` с настройками `Appium`

```
// src/test/java/base/Base.java
```

```
package base;

import io.appium.java_client.MobileElement;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.remote.MobileCapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.annotations.*;
import java.io.File;

public class Base {
    protected AndroidDriver<MobileElement> driver;

    @BeforeMethod
    public void setUp(){
        File appDir = new File("src/main/resources/apps");
        File app = new File(appDir, "ApiDemos-debug.apk");

        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability(MobileCapabilityType.DEVICE_NAME,
            "Android Emulator");
        capabilities.setCapability(MobileCapabilityType.PLATFORM_NAME,
            "android");
        capabilities.setCapability(MobileCapabilityType.AUTOMATION_NAME,
            "UiAutomator2");
        capabilities.setCapability(MobileCapabilityType.APP,
            app.getAbsolutePath());
    }
}
```

```

        capabilities.setCapability("avd", "Oreo");
        capabilities.setCapability("appPackage", "io.appium.android.apis");
        driver = new AndroidDriver<MobileElement>(capabilities);
    }

    @AfterMethod
    public void tearDown(){
        driver.quit();
    }
}

```

6. В примере 11.2 показан класс `HomePage` с методом, который извлекает текст из второго элемента на домашней странице, определяемого заданным идентификатором. В данном случае роль идентификатора играет значение атрибута `resource-id`.

Пример 11.2. Класс `HomePage`, выполняющий поиск элементов

```
// src/main/java/pages/HomePage.java
```

```

package pages;

import io.appium.java_client.MobileElement;
import io.appium.java_client.android.AndroidDriver;
import org.openqa.selenium.By;

public class HomePage{

    private AndroidDriver<MobileElement> driver;
    private By textItem = By.id("android:id/text1");

    public HomePage(AndroidDriver<MobileElement> driver) {
        this.driver = driver;
    }

    public String getFirstTextItem(){
        return driver.findElements(textItem).get(1).getText();
    }
}

```

7. В примере 11.3 показан класс `HomePageTest` с тестом, который открывает приложение и с помощью `TestNG` проверяет текст второго элемента на домашней странице.

Пример 11.3. Класс `HomePageTest` с тестом

```
// src/test/java/tests/HomePageTest.java
```

```

package tests;

import base.Base;
import org.testng.Assert;

```

```
import org.testng.annotations.Test;
import pages.HomePage;

public class HomePageTest extends Base {

    @Test
    public void verifyFirstTextItemOnHomePage() throws Exception {
        HomePage homePage = new HomePage(driver);
        Assert.assertEquals(homePage.getFirstTextItem(), "Accessibility");
    }
}
```

8. Тест можно запустить из IDE или из терминала, выполнив команду `mvn clean test`. Вы можете наблюдать выполнение теста в эмуляторе (если он не открыт, то Appium откроет его и затем запустит тест). Если тестирование запускается из командной строки, то в каталоге `/target/surefirereports/` автоматически будут созданы отчеты в формате HTML.

Тесты можно добавить в конвейер CI, чтобы организовать непрерывное тестирование вашего мобильного приложения. Если для автоматизации тестирования понадобятся дополнительные API, например имитирующие касание, прокрутку или листание, то обратитесь к официальной документации Appium (<https://oreil.ly/okAa5>).

Плагин Appium для визуального тестирования

Плагин визуального тестирования для Appium 2.0 использует OpenCV — инструмент обработки изображений с открытым исходным кодом — для сравнения изображений. В отличие от AppliTools Eyes, речь о котором шла в главе 6, у этого плагина меньше возможностей. Например, AppliTools Eyes может прокручивать страницу вниз и выполнять визуальное сравнение, не требуя писать программный код для этого, а с Appium нужно написать код, который получит и объединит несколько скриншотов, прежде чем передать результат для сравнения изображений. В то же время Appium имеет открытый исходный код, что позволяет без дополнительных затрат добавить хотя бы минимальный набор визуальных тестов к набору функциональных тестов Appium.

Давайте добавим несколько визуальных тестов к тесту пользовательского интерфейса, который мы создали ранее.

Установка и настройка

Чтобы установить и настроить плагин, выполните следующие действия.

1. Установите OpenCV, выполнив команду:

```
$ npm install -g opencv4nodejs
```

2. Установите плагин визуального тестирования Appium, используя команду:

```
$ appium plugin install images
```

3. Запустите сервер Appium командой:

```
$ appium server -ka 800 --use-plugins=images -pa /wd/hub
```

Рабочий процесс

Плагин `images` для Appium предоставляет две функции для визуального тестирования. Одна из них сравнивает базовое и реальное изображения:

```
SimilarityMatchingResult result =  
    driver.getImagesSimilarity(baselineImg, actualScreen, options);
```

Другая возвращает оценку сравнения из объекта `result`, которую можно использовать для объявления теста неудачным, если она меньше порогового значения, как показано здесь:

```
result.getScore() < 0.99
```

Оценка сравнения изменяется в диапазоне от 0 до 1. Оценка 1 соответствует идеальному совпадению, но из-за небольших различий результат не всегда получается идеальным, поэтому можно определить пороговое значение и с его помощью управлять чувствительностью теста в соответствии с потребностями проекта.

Рабочий процесс визуального тестирования с помощью этих двух функций прост: вы создаете набор базовых скриншотов приложения, сравниваете их с текущими скриншотами и объявляете тест неудачным, если оценка оказывается меньше порогового значения. В примере 11.4 показан ранее созданный тест пользовательского интерфейса Appium с дополнительными визуальными проверками. По умолчанию при первом запуске он создает базовые скриншоты. Обратите внимание на то, что вам нужно создать класс `BasePage` и добавить туда соответствующий код настройки.

Пример 11.4. Автоматизированное визуальное тестирование с помощью плагина Appium 2.0.

```
// src/main/java/pages/BasePage.java
```

```
package pages;  
  
import io.appium.java_client.MobileElement;  
import io.appium.java_client.imagecomparison.SimilarityMatchingOptions;  
import io.appium.java_client.imagecomparison.SimilarityMatchingResult;  
import org.openqa.selenium.OutputType;  
import io.appium.java_client.android.AndroidDriver;  
import java.io.File;  
import org.apache.commons.io.FileUtils;
```

```

public class BasePage {
    private File baselineDir = new File("src/main/resources/baseline_screenshots");

    public void checkVisualQuality(String screen_name,
        AndroidDriver<MobileElement> driver) throws Exception {
        File baselineImg = new File(baselineDir, screen_name + ".png");
        File actualScreen = driver.getScreenshotAs(OutputType.FILE);

        if (baselineImg.exists()) {
            SimilarityMatchingOptions options = new SimilarityMatchingOptions();
            options.setEnabledVisualization();
            SimilarityMatchingResult result =
                driver.getImagesSimilarity(baselineImg, actualScreen, options);
            if (result.getScore() < 0.99) {
                File imageDiff = new File("src/main/resources/baseline_screenshots"
                    + "FAIL_" + screen_name + ".png");
                result.storeVisualization(imageDiff);
                throw new Exception("Visual quality hampered");
            }
        } else {
            FileUtils.copyFile(actualScreen, baselineImg);
        }
    }
}

// src/test/java/tests/HomePageTest.java

public class HomePageTest extends Base {

    @Test
    public void verifyFirstTextItemOnHomePage() throws Exception {
        HomePage homePage = new HomePage(driver);
        Assert.assertEquals(homePage.getFirstTextItem(), "Accessibility");
        BasePage basePage = new BasePage();
        basePage.checkVisualQuality("home_page", driver);
    }
}

```

Плагин предоставляет функцию `result.storeVisualization()` для просмотра различий между двумя изображениями в случае неудачного теста. Чтобы увидеть, как работают тесты, сначала выполните команду `mvn clean test`. Она создаст базовый скриншот домашней страницы в папке `/src/main/resources/baseline_screenshots`. Если теперь повторно запустить тест, он должен выполниться успешно, потому что в приложении ничего не изменилось. Чтобы тест провалился, можете в качестве базового скриншота выбрать другой файл `.png` и запустить тест еще раз. Вы увидите, созданное в той же папке `baseline_screenshots` новое изображение, подчеркивающее различия (рис. 11.7).

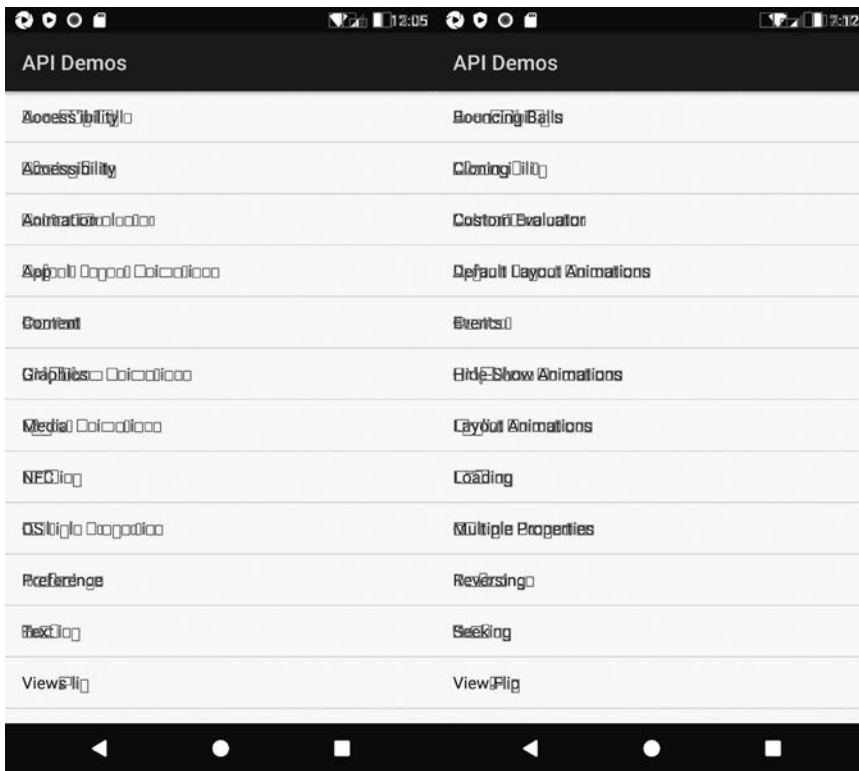


Рис. 11.7. Результат сравнения двух скриншотов в случае неудачи тестирования

К настоящему моменту мы шаг за шагом обсудили добавление автоматизированных тестов пользовательского интерфейса для проверки функционального поведения и визуального качества мобильного приложения с помощью Appium. Это два наиболее часто практикуемых типа тестов для мобильных приложений. Далее рассмотрим некоторые дополнительные инструменты, которые помогут реализовать другие типы тестирования.

Дополнительные инструменты тестирования

В этом разделе вы познакомитесь с некоторыми инструментами, которые можно использовать для тестирования производительности, безопасности, доступности и данных. Некоторые из этих типов тестирования сейчас редко практикуют в мобильном пространстве, тем не менее их желательно применять везде, где возможно, по тем же причинам, что и в веб-контексте.



Иллюстрации в этом разделе в основном показывают рабочий процесс применения инструментов в Android. Однако этот процесс вполне подходит и для iOS. Аналогичные инструменты для iOS были указаны в соответствующих подразделах при обсуждении стратегии тестирования мобильных приложений.

Database Inspector в Android Studio

Для исследования локальной базы данных мобильного приложения Android Studio предоставляет инструмент Database Inspector (<https://oreil.ly/Lf1vF>) с графическим интерфейсом. Как и любой другой клиент баз данных, его можно применять для добавления/редактирования/удаления данных и проверки поведения приложения.

Чтобы использовать Database Inspector, сделайте следующее.

1. Выберите в Android Studio пункт меню More Actions ► Profile or Debug APK (Дополнительные действия ► Профилирование или отладка APK) и затем файл .apk приложения. Обратите внимание на то, что для использования этого инструмента необходимо, чтобы приложение было собрано с отладочной информацией.
2. Выберите View ► Tools Window ► App Inspection (Вид ► Окно инструментов ► Исследование приложения). В нижней части экрана откроется панель проверки приложений.
3. Запустите приложение в эмуляторе Android Studio, щелкнув на зеленой кнопке запуска.
4. После этого Database Inspector откроется на панели исследования приложений. Обратите внимание: демонстрационное приложение не имеет локальной базы данных, поэтому на рис. 11.8 показан пример локальной базы данных другого приложения.

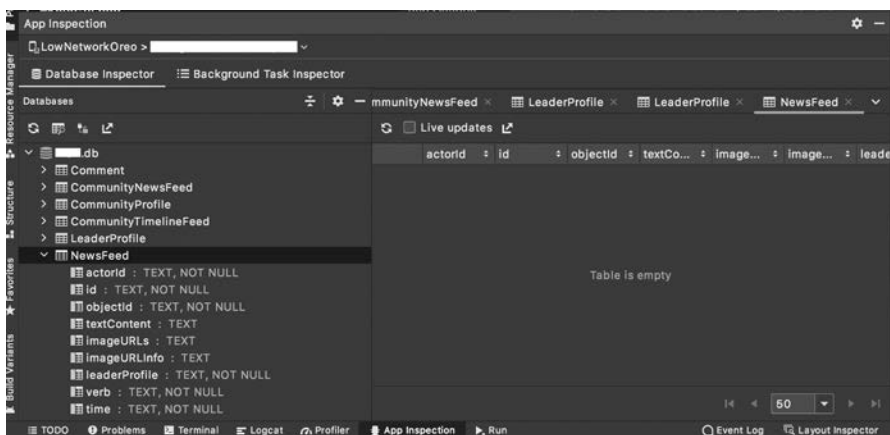


Рис. 11.8. Инструмент Database Inspector в Android Studio

В целом этот инструмент можно использовать, чтобы убедиться в сохранении данных, необходимых для работы приложения в автономном режиме, и в том, что конфиденциальная информация сохраняется только в зашифрованном виде.

Инструменты тестирования производительности

В подразделе «Тестирование производительности» раздела «Стратегия тестирования мобильных приложений» перечислены аспекты производительности мобильных приложений, которые необходимо тестировать. Здесь рассмотрим три инструмента, способные вам в этом помочь.

Monkey

Monkey (<https://oreil.ly/fp9oQ>) можно считать инструментом хаос-инжиниринга для Android-приложений. Он выполняет в пользовательском интерфейсе случайные последовательности действий, такие как касания, нажатия клавиш, щелчки и другие жесты, и сообщает о выявленных сбоях. Monkey — это простой инструмент командной строки. Если у вас уже установлена среда разработки Android Studio, то вы сможете провести стресс-тестирование демонстрационного приложения в эмуляторе или на физическом устройстве, выполнив следующую команду:

```
$ adb shell monkey -p "io.appium.android.apis" -v 2000
```

Эта команда отправит приложению 2000 различных событий, а вы в это время можете понаблюдать за его работой в эмуляторе или на устройстве. Если возникнут необработанные исключения или приложение перестанет откликаться, то Monkey остановит тестирование и сообщит о проблемах. При желании можно настроить стресс-тест для выполнения определенных событий, передав в команду соответствующие дополнительные параметры, как описывается в документации (<https://oreil.ly/fp9oQ>).

Расширенные возможности управления пропускной способностью сети

Еще один аспект тестирования производительности, который мы обсуждали ранее, — проверка работы приложения в разных сетевых условиях. Эмуляторы Android позволяют моделировать разные типы сетей, такие как GSM, GPRS, Edge, LTE и т. д. Дополнительно можно ограничить полосу пропускания, выбрав значение уровня сигнала: Good (Хороший), Moderate (Умеренный), Poor (Низкий), Great (Отличный) и т. д. Чтобы опробовать эту возможность, щелкните на кнопке **More options** (Дополнительные параметры) на боковой панели эмулятора и выберите **Cellular** (Сотовая связь) на панели настроек. Вы увидите параметры, доступные для

регулирования (рис. 11.9). Обратите внимание: здесь, помимо типа сети и уровня сигнала, есть и другие элементы управления, такие как состояние данных, состояние голоса и т. д. Можете использовать их для дальнейшей настройки сети в соответствии с требованиями тестового сценария.

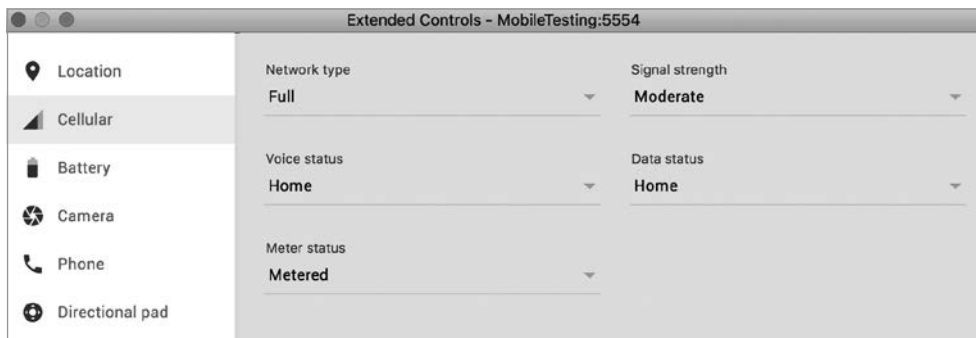


Рис. 11.9. Управление пропускной способностью сети в Android Emulator

Appium API для оценки производительности

Appium предоставляет функцию для оценки производительности приложения Android с точки зрения потребления памяти, процессора, батареи и сети. На ее основе можно создавать автоматизированные тесты производительности, дополняющие тесты пользовательского интерфейса, как показано далее:

```
driver.getPerformanceData("package_name", "perf_type", timeout);
```

где *package_name* — имя пакета тестируемого приложения; *perf_type* — компонент системы, потребление которого оценивается, например процессор, сеть и т. д.; *timeout* — количество секунд, в течение которых функция будет собирать данные о производительности, прежде чем выдать ошибку. Точное значение параметра *perf_type* можно получить вызовом другой функции Appium — `getSupportedPerformanceDataTypes()`. Сейчас поддерживаются четыре значения: `cpuinfo`, `memoryinfo`, `batteryinfo` и `networkinfo`.



Эта функция основана на инструменте командной строки Android `dumpsys`, который выводит диагностику системных служб. Соответственно, ее можно применять только с приложениями для Android.

С помощью этой функции можно добавить тест производительности, получить показатели производительности в разных точках пользовательского сценария, вы-

полняемого тестом пользовательского интерфейса, и проверять, находятся ли эти значения в определенных пределах. Например, можно сверить потребление памяти с пороговым значением после сложной операции в приложении. В примере 11.5 показаны данные о потреблении памяти сразу после открытия демонстрационного приложения для Android.

Пример 11.5. Вывод данных о потреблении памяти демонстрационным приложением с использованием Appium API для оценки производительности

```
driver.getPerformanceData("io.appium.android.apis","memoryinfo", 10);

// Вывод
[[totalPrivateDirty, nativePrivateDirty, dalvikPrivateDirty, eglPrivateDirty,
glPrivateDirty, totalPss, nativePss, dalvikPss, eglPss, glPss,
nativeHeapAllocatedSize, nativeHeapSize], [11432, 4708, 1692, null, null, 20807,
4926, 1717, null, null, 12648, 14336]]
```

Подробную информацию об интерпретации этих выходных значений и добавлении соответствующих утверждений для тестирования конкретного приложения вы найдете в документации `dumpsys` (<https://oreil.ly/qZ3wo>).

Инструменты тестирования безопасности

В этом разделе мы рассмотрим два инструмента для автоматизированного тестирования безопасности: MobSF и Qark.

MobSF

Как упоминалось ранее в этой главе, Mobile Security Framework — это инструмент с открытым исходным кодом, выполняющий статический и динамический анализ приложений для Android, iOS и Windows. Он также может помочь при анализе вредоносного ПО. Чтобы опробовать MobSF, выполните следующие действия.

1. Загрузите и установите Docker Desktop (<https://docs.docker.com/get-started>) и откройте приложение. (Для простого опробования Docker не нужно обладать обширными знаниями, но, устанавливая его на рабочий ноутбук, уточните политику своей компании относительно этого действия, потому что Docker бесплатен только для личного пользования.)
2. Запустите Docker-контейнер с MobSF, выполнив следующую команду:


```
$ docker run -it -p 8000:8000
opensecurity/mobile-security-framework-mobsf:latest
```
3. В результате MobSF будет установлен на вашем локальном компьютере. Откройте `http://0.0.0.0:8000` для просмотра веб-страницы MobSF.

4. Загрузите APK-файл демонстрационного приложения Android на этой странице. Также можете попробовать APK-файл InsecureBankv2 (<https://oreil.ly/YR5dX>), в учебных целях специально созданный с уязвимостями.
5. MobSF просканирует приложение и отобразит результаты на той же локальной веб-странице (рис. 11.10).

<div> MobSF </div> <div> RECENT SCANS STATIC ANALYZER DYNAMIC ANALYZER API DOCS DONATE ABOUT Search MD5 </div> <div> Static Analyzer </div> <div> Information Scan Options Signer Certificate Permissions Android API Browsable Activities Security Analysis </div>				
1	Debug Enabled For App [android:debuggable=true]	high	Debugging was enabled on the app which makes it easier for reverse engineers to hook a debugger to it. This allows dumping a stack trace and accessing debugging helper classes.	
2	Application Data can be Backed up [android:allowBackup=true]	medium	This flag allows anyone to backup your application data via adb. It allows users who have enabled USB debugging to copy application data off of the device.	
3	Activity (com.android.insecurebankv2.PostLogin) is not Protected. [android:exported=true]	high	An Activity is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device.	

Рис. 11.10. Результаты сканирования инструментом MobSF

Этот инструмент можно интегрировать в конвейер CI, как описано в документации (<https://oreil.ly/nm84B>), чтобы автоматически сканировать код при попытке коммита.

Qark

Qark (<https://github.com/linkedin/qark>) — еще один инструмент с открытым исходным кодом для тестирования безопасности приложений для Android. Он может сканировать исходный код или APK-файлы. Qark написан на Python. Можно установить его с помощью менеджера пакетов `pip`:

```
$ pip install qark
```

и запустить проверку безопасности APK с помощью команды:

```
$ qark --apk ~/path/to/apk --report-type html
```

Эта команда создаст файл отчета в формате HTML, в котором будут перечислены обнаруженные уязвимости.

Как упоминалось в главе 7, подобные инструменты автоматизированного сканирования безопасности помогают группам разработчиков программного обеспечения сместить тестирование безопасности на ранние этапы разработки. Однако в зависимости от навыков команды и контекста приложения вам все равно может потребоваться привлечь профессиональных тестировщиков ближе к концу разработки.

Accessibility Scanner

Accessibility Scanner — это инструмент тестирования доступности приложений для Android, который можно установить из Google Play (<https://oreil.ly/zSNKd>). Установив его на свое устройство и предоставив необходимые разрешения, вы сможете запустить приложение, которое хотите протестировать, и нажать синюю кнопку с галочкой, чтобы начать сканирование доступности. Затем появится возможность записать поток работы приложения (рис. 11.11).

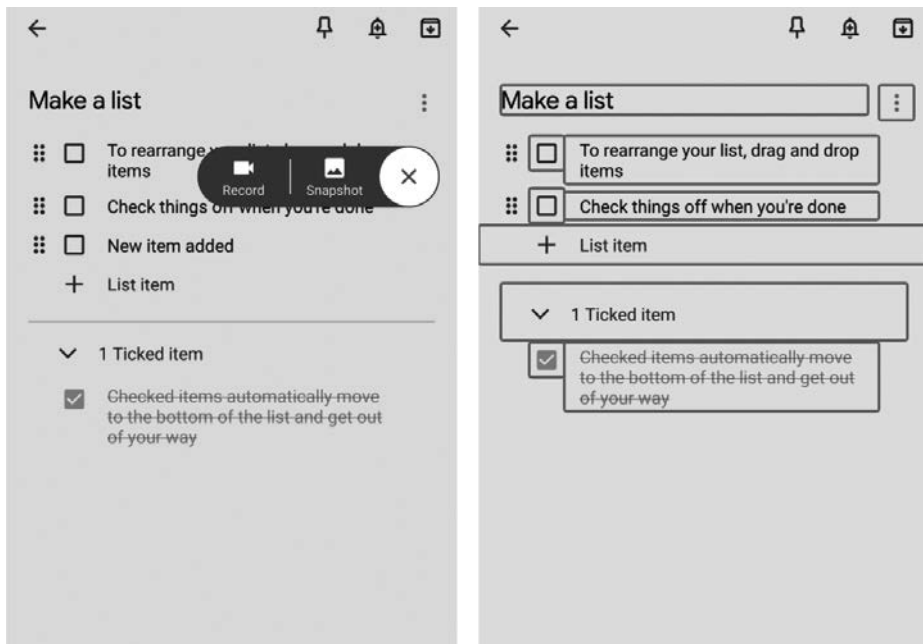


Рис. 11.11. Результаты тестирования приложения с помощью Accessibility Scanner

Попробуйте прямо сейчас — откройте любое приложение на устройстве Android и начните запись. Перемещайтесь по приложению и, закончив, остановите запись еще одним нажатием на синюю кнопку с галочкой. Затем приложение Accessibility Scanner покажет результаты тестирования доступности для всех экранов, по которым вы перемещались, выделяя элементы, доступность которых можно улучшить (см. рис. 11.11). При нажатии на выделенный текст отображается описание проблемы. Это приложение можно использовать уже на этапе разработки для выявления недостающих специальных возможностей.

Поздравляю! Вы успешно освоили широкий спектр инструментов для тестирования различных функциональных и межфункциональных требований мобильных приложений!

Перспективы — пирамида тестирования мобильных приложений

Теперь, познакомившись с различными инструментами автоматизированного тестирования мобильных приложений, давайте посмотрим, как они сочетаются друг с другом и соответствуют ли обычной пирамиде тестирования.

Как упоминалось ранее, архитектура типичного мобильного приложения аналогична архитектуре веб-приложения: приложение вызывает сервисы, которые, в свою очередь, взаимодействуют с БД. Таким образом, тесты на нижних уровнях соответствуют пирамиде тестирования. Как вы помните, она должна иметь широкое основание, содержащее тесты микроуровня, и узкую верхушку с тестами макроуровня. Однако некоторые специалисты утверждают, что в тестировании мобильных приложений невозможно достичь формы пирамиды. Они говорят, что, по их опыту, пирамида тестов мобильных приложений имеет перевернутую форму с широким слоем тестов пользовательского интерфейса, менее широким — ручных тестов и узким — модульных тестов. Их опыт в основном свидетельствует об ограниченной применимости модульных тестов и тестов пользовательского интерфейса в контексте мобильных приложений.

С помощью модульных тестов они, как обычно, проверяют небольшие фрагменты функциональности, предоставляемые классом или методом. При этом считают, что можно не писать модульные тесты для проверки поведения функций, зависящих от системных функций, так как подразумевается, что производитель ОС сам будет тестировать системные функции. Такое мнение предполагает необходимость обширного сквозного тестирования, ручного или автоматизированного, чтобы убедиться, что системные функции работают должным образом в контексте приложения на разных типах устройств. Кроме того, функции, зависящие от аппаратного обеспечения, такие как связь с камерой, датчиками и т. д., а также функции, связанные с удобством применения, такие как простота прокрутки, расширенные жесты и т. д., не могут быть тщательно протестированы ни в рамках модульных тестов, ни в рамках тестов пользовательского интерфейса. Эти аспекты требуют усилий по ручному тестированию.

Учитывая ограниченную применимость модульных тестов и тестов пользовательского интерфейса, форма пирамиды тестирования мобильных приложений определяется характеристиками приложения. Для приложений с обширной функциональной логикой и мало зависящих от внешних факторов, таких как аппаратное обеспечение устройства и системные функции, пирамида мобильных тестов будет выглядеть похожей на традиционную пирамиду тестирования из-за большого количества тестов более низкого уровня, охватывающих функциональную логику. Для приложений с небольшим объемом функциональной логики, но сильной зависимостью от внешних факторов пирамида тестирования может приобрести

перевернутый вид. В таких случаях вам придется планировать дополнительные возможности тестирования и постараться достичь баланса между написанием большего количества автоматизированных тестов пользовательского интерфейса и исчерпывающим ручным регрессионным тестированием.

Ключевые выводы

- Учитывая растущую зависимость людей от мобильной сферы и прибыль, которую она приносит бизнесу, можно ожидать, что в ближайшие годы мы станем свидетелями развития тенденции проникновения мобильных приложений во все сферы жизни. Поэтому, как разработчики и тестировщики ПО, мы должны подготовиться к этому и приобрести соответствующие навыки тестирования.
- Тестирование мобильных приложений отличается от тестирования веб-приложений. В этой главе мы рассмотрели некоторые нюансы мобильной среды с трех разных точек зрения и сосредоточились на аспектах, связанных с устройствами, приложениями и сетью. Разнообразие в каждой из этих областей становится серьезной проблемой для разработчиков ПО на всех этапах разработки: при проектировании, в ходе собственно разработки и во время тестирования.
- Ваша стратегия тестирования мобильных приложений должна придерживаться основ фулстек-тестирования и уделять особое внимание тестированию на микро- и макроуровне, практике раннего тестирования и проверке различных показателей качества, таких как безопасность, производительность и доступность.
- В разделах с упражнениями и обсуждением дополнительных инструментов тестирования было представлено множество средств функционального и межфункционального ручного и автоматизированного тестирования, которые можно использовать для проверки мобильных приложений.
- Форма пирамиды тестирования мобильных приложений может меняться в зависимости от особенностей приложения. Этот важный аспект следует принять во внимание как можно раньше, чтобы запланировать время и средства для тестирования.

ГЛАВА 12

За рамками тестирования

Практики следуют указаниям, эксперты понимают принципы!

К настоящему моменту мы обсудили все навыки тестирования, необходимые для успешного создания высококачественных мобильных и веб-приложений. Мы выяснили, что тестирование — это обширное пространство, развивающееся уже на протяжении нескольких десятилетий и включающее в себя все новые процессы, инструменты и методологии. Сегодня существует десять основных навыков фулстек-тестирования (были описаны в предыдущих главах), а завтра их может стать намного больше. Однако даже в такой динамичной среде основополагающие принципы тестирования остаются неизменными независимо от технологии или области применения. Понимание этих принципов послужит фундаментом для достижения успеха независимо от того, как в каких направлениях продолжит расширяться пространство тестирования в будущем.

В этой главе я представлю краткий обзор основных принципов тестирования и их важнейших преимуществ, а также расскажу, как на основе этих принципов развивались существующие ныне инструменты и практики. Затем мы рассмотрим, как навыки программирования каждого отдельного человека вкупе с его техническими навыками способствуют общему успеху команды при разработке высококачественного ПО.

Основные принципы тестирования

На рис. 12.1 показаны семь основных принципов тестирования. Мы подробно исследуем их в дальнейшем.

Предотвращение дефектов вместо обнаружения

Тестирование в основном направлено на поиск проблем в приложении, однако предотвращение проблем тоже следует рассматривать как одну из его основ. Главная причина того, почему следует стараться предотвращать дефекты, заключается

в стоимости их исправления. Это можно сравнить с замазыванием трещины на гладко окрашенной стене и ее закрашиванием — иногда только что окрашенный участок сильно выделяется на общем фоне и приходится красить всю стену заново! Точно так же дефекты в ПО могут привести к значительным изменениям в архитектуре, требующим огромного количества переделок и больших затрат. Следовательно, этот основной принцип предполагает принятие практик, инструментов и методов, позволяющих в первую очередь предотвращать появление дефектов, а не обнаруживать и устранять их.



Рис. 12.1. Основные принципы тестирования

Далее перечислены некоторые современные практики, направленные на реализацию этого принципа.

- Совещания по планированию итераций (iteration planning meetings, IPM), которые проводятся в начале итерации или спринта для подробного обсуждения пользовательских историй. IPM — это открытое пространство, где команды могут провести мозговой штурм для выявления недостающих связей и крайних случаев в пользовательских историях.
- Процесс «Три амиго» (<https://oreil.ly/9v2hs>), в ходе которого представители бизнеса, разработчики и тестировщики тщательно обдумывают каждую функцию

на этапе анализа. Этот процесс направлен на сбор точек зрения всех трех ролей, чтобы не упустить из виду интеграцию, крайние случаи и другие бизнес-требования.

- Аналогично запуск пользовательской истории предполагает повторение процесса «Три амиго» непосредственно перед началом разработки пользовательской истории. Это тоже обычная практика в раннем тестировании, позволяющая тестировщикам фиксировать и обсуждать тестовые сценарии в начале обсуждения истории.
- Записи об архитектурных решениях (architecture decision records, ADR) (<https://oreil.ly/qSMX4>) и стратегии тестирования обсуждаются и документируются, чтобы служить ориентиром на пути к общим целям создания качественного проекта.
- Практика разработки через тестирование (test-driven development, TDD) требует обдумывания крайних случаев даже для небольших фрагментов кода.
- Парное программирование — еще одна практика разработки, направленная на предотвращение пропуска крайних случаев и появления плохого кода, приводящего к дефектам.
- Аналогично программы-линтеры выявляют дефекты в коде уже в процессе его создания разработчиком.

Как видите, существует множество практик, направленных на предотвращение дефектов, и их можно применять к любой новой области, например к такой, как данные.

Эмпатическое тестирование

Суть тестирования заключается в том, чтобы в конце концов поставить себя на место конечного пользователя. Приняв на себя роль тестировщиков, мы должны исходить из его интересов и определить, какие факторы и детали технической реализации мешают удовлетворению важных бизнес-требований. Мы не можем ограничиться простой проверкой критериев приемлемости пользовательской истории и двигаться дальше — мы должны попробовать применить приложение так, как это будет делать типичный конечный пользователь. Поэтому, прежде чем приступить к тестированию, важно понять переживания различных типов конечных пользователей, на которых ориентировано приложение. Часто команды жертвуют их потребностями, оправдывая это сложностью разработки и жесткими сроками. Однако, играя роль тестировщика, мы в первую очередь должны встать на точку зрения конечного пользователя и обсуждать компромиссы, принимая во внимание его интересы. Несмотря на то что мы постоянно работаем со своими командами, при тестировании должны ставить на первое место конечных пользователей.

Тестирование на микро- и макроуровне

Как обсуждалось в главе 1, чтобы получить высококачественное ПО, его нужно тестировать как на микро-, так и на макроуровне. Напомню, что тестирование на микроуровне предполагает выделение небольшой части функциональности и ее детальное тестирование — например, тестирование расчета общей суммы заказа с различными граничными условиями (отрицательные цены, длинные десятичные дроби и т. д.). Тестирование на макроуровне предполагает более широкий подход с охватом функциональных потоков, распространения данных между модулями, интеграции компонентов и т. д. Например, тесты на макроуровне могут быть сосредоточены на тестировании процесса создания заказов, связи со сторонними сервисами, потоков пользовательского интерфейса, сбоев при создании заказов и т. д.

В главе 3 мы видели продуманную стратегию автоматизированного функционального тестирования с различными типами тестов на микро- и макроуровне. Согласно этой стратегии модульные, интеграционные и контрактные тесты сосредоточены на микроуровне, тогда как тесты API, функциональные тесты пользовательского интерфейса, визуальные тесты и т. д. — на макроуровне. Мы также обсудили, как дисбаланс в распределении тестирования на микро- и макроуровне может задерживать обратную связь и приводить к снижению качества.

Еще одно критическое следствие такого дисбаланса — выявление проблем уже в процессе эксплуатации. Такое случается, потому что команды, фокусирующиеся только на тестировании на макроуровне, зачастую упускают из виду детали. Например, они могут протестировать сценарии макроуровня, такие как успешное создание заказа и сбой при его создании из-за недоступности товара. Но когда в промышленной среде цены на товары оказываются отрицательными или имеют неожиданное количество десятичных знаков, создание заказа может завершиться неудачей. Поэтому во время тестирования крайне важно постоянно увеличивать и уменьшать масштаб, чтобы получить как общую, так и детализированную картину.

Быстрая обратная связь

Этот принцип заключается в раннем выявлении дефектов, чтобы цикл исправления и, как следствие, цикл выпуска могли протекать быстрее. Существует заметная корреляция между временем устранения дефекта, и тем, насколько поздно он выявлен. Когда функция находится в разработке, программист прекрасно помнит контекст кода и способен быстро понять основные причины ошибок и исправить их. Но, когда он переходит к другим функциям, а база кода продолжает каждый день расти за счет рефакторинга, этот контекст исчезает и выяснение и устранение причин становится более длительным и дорогостоящим процессом.

Более того, удлинение цикла отслеживания дефектов в значительной степени способствует задержке цикла исправления. Например, представьте себе высокоприоритетную ошибку, обнаруженную через две недели после разработки функции. Вам нужно время, чтобы создать карточки ошибок, отсортировать их, отыскать подходящего разработчика, который сможет их исправить. Эти задачи могут занять дни и даже недели! И после всех этих задержек в некоторых худших сценариях может обнаружиться, что невозможно исправить дефект без серьезного рефакторинга кода, разработанного в промежуточный период, что еще больше задерживает выпуск. Из-за такого роста цены исправления дефектов желательно сосредоточиться на создании более быстрых циклов обратной связи.

Итак, насколько рано следует начинать тестировать фрагмент кода, чтобы обеспечить быструю обратную связь? Ускорить получение обратной связи позволяет раннее тестирование, и в предыдущих главах мы не раз видели, как это реализовать. Только чтобы помочь вам вспомнить практики, обсуждавшиеся в главах 2–4, которые позволяют получить более быструю обратную связь, напомним, что вы можете реализовать тестирование в среде разработки (запуск автоматических тестов на машине разработчика) и пирамиду тестирования. Кроме того, быстрее получать обратную связь по недостающим бизнес-функциям вам помогут одобрение пользовательских историй владельцами продуктов или представителями бизнеса и демонстрация сделанного всем заинтересованным сторонам после завершения каждого спринта.

Проще говоря, тестирование для быстрого получения обратной связи эквивалентно сбору урожая в нужное время. При затягивании сроков приходится довольствоваться менее качественным урожаем.

Постоянная обратная связь

Быстрота обратной связи должна подкрепляться ее постоянством. Недостаточно протестировать функцию один раз, а затем забыть о ней до выпуска. Необходимо продолжить регрессионное тестирование, чтобы получать обратную связь о сохранении работоспособности текущей функции, интеграции функций по мере разработки новых и рефакторинга существующего кода. Такие механизмы постоянной обратной связи помогают выявлять проблемы, когда они еще относительно малы, и предотвращать нарушение сроков выпуска. Постоянная обратная связь также дает команде возможность обеспечить непрерывную доставку!

Как обсуждалось в главе 4, основным способом получения постоянной обратной связи является внедрение методов непрерывного тестирования. Например, все функциональные тесты на микро- и макроуровне, а также тесты межфункциональных требований должны внедряться в конвейер CI и выполняться для каждого коммита. Это обеспечит постоянную обратную связь по всем аспектам качества и позволит команде осуществлять непрерывную доставку.

Измерение показателей качества

Все, что измеряется, имеет тенденцию улучшаться! Цель наличия показателей качества (KPI) в любой области — их отслеживание и итеративное улучшение. Поэтому, пытаясь получить высококачественные результаты, мы должны измерять качество. Тем не менее, когда показателям уделяется чрезмерно большое внимание, члены команды склонны искать способы их подтасовки, забывая о конечной цели. По этой причине следует разумно использовать показатели качества и ориентировать команду на достижение общих целей в области качества.

Далее перечислены некоторые показатели качества, регулярное слежение за которыми принесет пользу команде.

Дефекты, обнаруженные автоматизированными тестами на всех уровнях

Автоматизированные тесты порождают у команды ощущение безопасности, и когда большинство дефектов обнаруживается на ранних стадиях, разработчики чувствуют себя увереннее при внесении новых изменений. Этот показатель отражает также силу системы безопасности.

Время от коммита до развертывания

Как мы видели ранее, быстрая обратная связь имеет решающее значение для ускоренного продвижения вперед. Когда разработчик фиксирует изменения, их нужно немедленно проверять автоматизированными тестами в конвейере CI и развертывать в среде контроля качества, чтобы можно было начать исследовательское тестирование. Я видела команды, конвейеры CI которых требовали много времени для создания «зеленой» сборки из-за нестабильных тестов и проблем со средой, что приводило к задержке обратной связи и снижению производительности.

Количество автоматических развертываний в средах тестирования

Этот и предыдущий показатели говорят о том, насколько быстро и успешно команда способна вносить изменения. В идеале она должна иметь хорошую систему безопасности, обеспечивающую быстрое и стабильное развертывание. Если обнаружится, что количество автоматических развертываний в средах тестирования невелико из-за сбоев инфраструктуры тестирования или по другим причинам, значит, цикл обратной связи нуждается в улучшении.

Дефекты регрессионного тестирования, обнаруженные во время тестирования пользовательских историй

Дефекты регрессионного тестирования, обнаруженные на этапе тестирования пользовательских историй, указывают на отсутствие сценариев применения в бизнесе или отсутствие автоматизированных тестов. Например, автоматизированные тесты в CI пропустят замену `like` на `equals` в SQL-запросе, если тестовые данные были разработаны так, чтобы соответствовать обоим вариантам запроса.

Как обсуждалось в главе 3, если во время тестирования пользовательских историй обнаруживаются дефекты регрессионного тестирования, это может служить признаком того, что команды следуют антишаблонам в автоматизированном тестировании. В таких случаях им стоит задуматься о коренных причинах этих дефектов и регулярно совершенствовать свои процессы.

Охват автоматизацией в зависимости от серьезности тестовых сценариев

Ведите подробный учет охвата кода автоматизированными тестами, чтобы избежать отставания. Отслеживание этого показателя поможет заранее спланировать итерации, чтобы ликвидировать отставание, если таковое имеется.

Дефекты в промышленной среде и их серьезность

Отслеживание дефектов в промышленной среде позволяет получить более полное представление о невозможных вариантах использования, не существующей конфигурации, несоответствиях в данных и любых других проблемах, которые команда могла упустить из виду. Определите основные их причины и автоматизируйте тестирование. Кроме того, разработайте стратегию тестирования и продолжайте развивать ее по мере того, как приложение и команда выходят на новые горизонты.

Оценка удобства применения конечными пользователями

Собирайте отзывы конечных пользователей об общем удобстве на этапе разработки. Это поможет улучшить показатели, связанные с организацией пользовательского интерфейса (например, уменьшить количество щелчков кнопкой мыши для получения информации, применять текст наряду со значками и т. д.).

Сбои из-за проблем с инфраструктурой

Отслеживайте проблемы с инфраструктурой, такие как периодические отключения сервисов в средах тестирования, проблемы в конвейерах CI, несоответствия в конфигурациях сред тестирования и разработки и т. д. Иногда для поддержания масштабируемости и стабильности кода инфраструктуры может понадобиться погасить технический долг.

Показатели, относящиеся к межфункциональным аспектам

Последовательно измеряйте ключевые показатели эффективности и демонстрируйте результаты своим командам. Включите статистику по автоматизированным тестам безопасности и уязвимостям, обнаруженным во время автоматизированного сканирования, в итоговую информацию по итерациям. Также включите в нее показатели охвата автоматизацией тестирования межфункциональных требований (охват тестирования с разными версиями браузеров, результаты хаос-инжиниринга, охват тестирования локализации и т. д.).

Многие из упомянутых здесь показателей связаны с четырьмя ключевыми показателями, обсуждавшимися в главе 4, которые измеряют качество с точки зрения стабильности кода и темпа разработки, характерного для команды. Как вы наверняка помните, один из четырех ключевых показателей — *время выполнения заказа* (время от коммита кода до его готовности к промышленному развертыванию) для элитной команды составляет менее одного дня. При хорошем охвате кода автоматизированными тестами команда будет увереннее вносить такие быстрые изменения.

Точно так же в элитной команде показатель *частоты развертывания* должен иметь значение «по требованию». Измеряя время от принятия решения до развертывания, а также количество развертываний в тестовых средах в течение дня, мы получаем представление о темпах разработки. Дефекты в промышленной среде расскажут нам о *проценте неудачных изменений* (процент неудачных изменений, попавших в промышленную среду), который для высококачественного продукта должен составлять 0–15 %. Последовательно отслеживая и обсуждая эти показатели, команда будет неуклонно приближаться к своей главной цели — созданию высококачественного программного обеспечения.

Общение и сотрудничество: ключ к высокому качеству

Тестирование не может проводиться в отрыве от прочей деятельности. Чтобы оно приносило пользу, необходимо правильное информирование о бизнес-требованиях, знании предметной области, технической реализации, деталях среды и т. д. Это требует последовательного сотрудничества все членов команды и взаимодействия между ними. Общение может осуществляться в процессе организационных мероприятий, таких как выступления, презентации историй, встречи для планирования итераций, в ходе тестирования в среде разработки и посредством различной документации, такой как карты историй, записи архитектурных решений, стратегии тестирования, отчеты об охвате тестированием и т. д. В современном мире, когда члены распределенных команд работают в разных часовых поясах, синхронное общение не всегда возможно, поэтому мы должны стараться организовать своевременную передачу информации с помощью надлежащей документации и асинхронных средств, таких как видеозаписи и электронные письма.

Следование этим семи основным принципам поможет командам разработчиков ПО создавать эффективные стратегии тестирования, даже когда они осваивают новые области технологического пространства. Я сама применяла эти принципы в проектах с новыми стеками технологий и в незнакомых областях и видела, как они дают превосходные результаты.

Навыки межличностного общения помогают формировать мышление, ориентированное на качество

Важно еще раз подчеркнуть, что создавать высококачественное программное обеспечение помогают несколько аспектов разработки ПО: проектирование, анализ, разработка, инфраструктура и т. д. Тестирование качества — один из таких аспектов, причем очень важный. Поэтому все члены команды должны вместе работать над достижением высокого качества. Ни один из них не может самостоятельно обеспечить необходимый уровень качества, и никто в команде *не должен* оставаться в стороне. Борьба за качество — это как эстафета: команда не может выиграть забег, если хотя бы один спортсмен будет бежать медленно. И навыки межличностного общения играют решающую роль в формировании в команде мышления, ориентированного на качество. Если вы профессионально занимаетесь тестированием или отвечаете за тестирование в своей команде, то следующие навыки межличностного общения помогут вам при формировании в ней коллективного мышления, ориентированного на качество.

Способность добиваться результатов

Если каждая роль в команде будет сосредоточена на достижении качественных результатов, то и весь коллектив будет настроен на обеспечение высокого качества. Например, за разработку интуитивно понятного пользовательского сценария отвечает дизайнер пользовательского интерфейса, за создание удобного для клиента продукта — владельцы продукта и бизнес, а за создание хорошей архитектуры и надежного кода — разработчики. Аналогично тестировщики в первую очередь отвечают за тестирование и должны побуждать команду включать его в свою повседневную практику. Например, они отвечают за применение методов и инструментов, предотвращающих дефекты, за соблюдение практики непрерывного тестирования, отслеживание уровня автоматизации каждой пользовательской истории, а также за другие методы, описанные в книге.

Сотрудничество

Привить представление о том, что качество является обязанностью команды, можно только при тесном сотрудничестве со всеми ее членами, клиентами или заинтересованными представителями бизнеса. Будучи негибкими в своих устремлениях и безразличными к общению с другими членами команды, мы не добьемся качественных результатов. Например, выработка стратегии тестирования совместно с разработчиками поможет достичь этой цели, так же как сотрудничество с представителями бизнеса при выявлении недостающих тестовых сценариев поможет предотвратить возникновение дефектов.

Эффективное общение

Иногда от того, как мы общаемся, зависит, будет задача успешно выполнена или нет. Эффективное общение также означает выбор подходящих средств

и времени для общения. В частности, тестировщики должны регулярно и четко сообщать команде об общем качестве продукта и о том, что необходимо для достижения желаемого уровня качества.

Расстановка приоритетов

Тестирование может растянуться до бесконечности, если своевременно не расставить приоритеты. Иногда задачи, кажущиеся небольшими с точки зрения разработки, требуют значительных незапланированных усилий по тестированию, что приводит к нарушению планов работ. Чтобы избежать таких ситуаций, тестировщики должны заранее расставить приоритеты действий по тестированию для каждой пользовательской истории и обеспечить учет требуемых усилий в рамках итерации. Это позволит команде успешно реализовывать функции без ущерба для качества.

Управление заинтересованными сторонами

К заинтересованным сторонам проекта относятся клиенты, менеджеры, товарищи по команде, технические руководители и все остальные, кто может влиять на наши действия. Мы должны постоянно управлять ожиданиями заинтересованных сторон в отношении качества. Клиенты могут ожидать стопроцентного охвата автоматизацией, что не всегда реально, а менеджеры могут быть больше заинтересованы в соблюдении сроков выпуска, чем в качестве. Управление этими ожиданиями и помощь в их формировании посредством сотрудничества, эффективного общения и определения приоритетов приведут к коллективному успеху.

Обучение/наставничество

Привлечение новых членов — обычное явление в командах, и мы не можем ожидать, что новички с самого начала будут знать все методы и инструменты, которыми пользуется команда. Однако, придерживаясь того, что качество — это коллективная ответственность команды, каждый ее член должен поддерживать общее мнение относительно методов и инструментов тестирования. Поэтому и мы, как тестировщики (наряду со всеми другими ролями в команде), должны делиться своими знаниями с новыми членами команды и помогать им быстрее влиться в работу.

Кроме того, наставничество предполагает не только адаптацию новых членов в рамках проекта, но и постоянное обучение и совершенствование подопечных, особенно в области межличностного общения, чтобы впоследствии они сами могли выступать в роли лидеров по качеству в команде.

Влияние

Влияние важно, особенно в работе с большими командами и новыми клиентами. Без этого, даже разработав разумную стратегию тестирования, мы не сможем реализовать ее по всем направлениям так, как нам хотелось бы. Влияние имеет решающее значение для поддержки стратегии тестирования и убеждения

заинтересованных сторон вкладывать средства в новые инструменты и методы тестирования. Конечно, не существует единого рецепта укрепления влияния, но способность последовательно добиваться высококачественных результатов, а также владение шестью навыками межличностного общения, перечисленными ранее, помогут вам в достижении этой цели!

Навыки межличностного общения сложнее освоить, чем технические приемы и методы, и для их совершенствования нужна ежедневная практика. Но по мере улучшения этих навыков, вы, к своему удивлению, можете заметить, что уже неплохо владеете некоторыми из них, а используя их должным образом, обнаружите, что они очень полезны для достижения успеха и вами, и всей командой.

Заключение

Мы подошли к концу обширного исследования навыков тестирования, необходимых для создания высококачественных мобильных и веб-приложений. Теперь я должна отметить, что тестирование — это процесс непрерывного обучения. Активно практикуя все, что мы здесь обсудили, вы будете продолжать развивать свои умения. Кроме того, как я отметила в начале, тестирование — это быстро развивающаяся область, в которой постоянно появляются новые инструменты, процессы и приемы. Такой быстрый рост временами может показаться ошеломляющим. В этих случаях я советую приостановиться и вспомнить, что все новые разработки, по сути, соответствуют одному из основных принципов, а понять, какому именно, можно, лишь делая небольшие шаги. Даже простое сочетание навыков фулстек-тестирования с навыками межличностного общения поможет вам эффективно создавать высококачественное программное обеспечение!

Итак, мы подошли к концу книги. Далее следует дополнительная глава, в которой обсуждаются четыре новые технологии и некоторые характерные для них аспекты тестирования. В конце концов, цель этой книги состояла в том, чтобы помочь читателю мыслить более широко, не ограничиваясь рамками мобильных и веб-приложений.

Пока вы еще здесь, я хотела бы поблагодарить вас за то, что прошли этот долгий путь вместе со мной. Вы показали свою приверженность созданию высококачественного программного обеспечения, что действительно заслуживает похвалы! Я надеюсь, что книга стала для вас эффективным руководством по освоению новых навыков тестирования и пролила свет на современные методы тестирования, которые вы можете с пользой применять в своей работе. До новых встреч в мире тестирования, всего наилучшего и спасибо за предоставленную мне возможность путешествовать вместе с вами по этой книге! :)

Введение в тестирование новых технологий

Быстрые изменения в технологиях могут
быть волнующими и головокружительными
одновременно!

За последнее десятилетие технологии сделали гигантский шаг вперед. Многие из того, что мы видели в детстве в научно-фантастических фильмах, сегодня стало явью: беспилотные летательные аппараты, авторизация по отпечаткам пальцев, умные помощники, видеоигры с полным погружением, и это далеко не полный список. У нас на слуху так много новых модных словечек: искусственный интеллект, машинное обучение, блокчейн, дополненная реальность, виртуальная реальность, смешанная реальность, боты и многое другое! Трудно даже запомнить их все сразу. Один из способов освоить такое огромное разнообразие технологий — сгруппировать их по темам, например так.

Человекоподобные взаимодействия

Долго для взаимодействия с компьютером нам хватало мыши и клавиатуры. В современном мире к ним добавились возможность прикосновения, голосовое управление, жесты и многие другие. Сейчас с нами взаимодействуют, а точнее, разговаривают Fitbit, Alexa, Алиса!

Дополненный интеллект

Технологии дополненного интеллекта делают нашу жизнь намного проще. Умные помощники, персонализированные рекомендации и чат-боты — вот лишь несколько примеров того, как технологии безвозвратно изменили ее.

Платформы как стандарты

Текущая тенденция в развитии технологий заключается в формировании технологических платформ (<https://oreil.ly/SEKEk>), абстрагирующих данные, сервисы, инфраструктуру и многое другое с целью обеспечить возможность повторного применения и масштабирования. Она позволяет постоянно внедрять новые продукты, учитывающие потребности рынка. Так называемые *суперприложения*

(<https://oreil.ly/an6sR>), такие как Uber, WeChat, Grab и Gojek, используют платформы в качестве основы.

Устройства, подключенные к Интернету

Давайте на мгновение перестанем думать о людях. Теперь у нас есть вещи, подключенные к Интернету! Мы живем в мире, где друг с другом общаются наши телефоны, часы и кофемашины.

Подкаст *Seismic Shifts* компании Thoughtworks (<https://oreil.ly/ijGuG>) и отчет *Looking Glass* (<https://oreil.ly/V6IjS>) представляют подробные обзоры технологических достижений для тех, кто желает изучить их подробнее.

Многие из этих технологий еще не заняли господствующего положения, следовательно, навыки их тестирования необязательны, тем не менее разумно подготовиться к этому заранее. Цель этой главы — дать краткое введение в четыре новые технологии — ИИ/МО, ДР/ВР, блокчейн и Интернет вещей — и обсудить аспекты тестирования каждой из них. Очевидно, что каждая из этих тем заслуживает отдельной книги, поэтому в данной главе я дам лишь краткий обзор этих технологий и отмечу направление, в котором они движутся.

Искусственный интеллект и машинное обучение

Искусственный интеллект (ИИ) — это область информатики, посвященная использованию машин для решения задач, которыми обычно занимаются люди, путем имитации человеческого интеллекта. В частности, *общий ИИ* — это теоретическая конструкция, способная делать все, что может человек. ИИ реализуется посредством машинного обучения (МО) — еще одной области информатики, основанной на идее возможности запрограммировать компьютеры на обучение на основе опыта, а не на выполнение вычислений по четко установленному алгоритму.

Термины ИИ и МО часто используются как взаимозаменяемые. Однако это несколько разные области: любую программу, которая демонстрирует человеческое поведение, можно назвать искусственным интеллектом, но если она формирует свое поведение не на основе изучения опыта, то есть не на основе исторических данных, то это не машинное обучение. Это различие станет понятнее, когда мы поговорим о подходе к программированию с использованием машинного обучения.

Введение в машинное обучение

Обычно, разрабатывая приложение, мы пишем последовательность инструкций, которые компьютер должен выполнить. По крайней мере до сих пор мы были уверены, что именно так они работают. Но новость о том, что компьютеры могут учиться на своем опыте без явного программирования, выглядит чрезвычайно

интригующей. Чтобы прояснить, что это значит, рассмотрим пример — фильтр оскорбительного контента в приложении для социальных сетей. Он поможет нам лучше понять разницу между традиционным подходом к программированию и машинным обучением.

Чтобы создать фильтр оскорбительного контента традиционным способом, мы должны начать с перечисления критериев, идентифицирующих контент как оскорбительный, описать их как правила в форме программного кода и удалить сообщения, подпадающие под эти правила. Например, можно написать код, проверяющий присутствие слов из чек-листа, таких как «самоубийство», «секс», нецензурные слова и т. д. Аналогично можно проверять идентификаторы пользователей, известных своей склонностью к оскорблениям, отмечать их контент как оскорбительный и автоматически отбрасывать его.

Но достаточно ли этого? После того как мы определим правила в коде, использующие список контрольных слов, злоумышленники быстро вводят новые слова, чтобы обойти запреты. Аналогично, когда накладываются ограничения на существующие учетные записи, они создают новые для отправки контента. В таком предметном пространстве, где сами правила недетерминированы, разработка надежного решения с использованием традиционного подхода к программированию — довольно сложная задача. Именно здесь на помощь приходит машинное обучение.

Используя подход к программированию на основе МО (рис. 13.1), мы вводим в модель машинного обучения огромное количество исторических данных, маркированных как оскорбительные или неоскорбительные. Это называется *обучением модели*. По сути, модель — это математический алгоритм, изучающий различия между двумя типами контента на основе данных. В некотором смысле это похоже на то, как учится человеческий мозг. С годами мы видим множество яблок разных размера, формы и цвета под разными углами и мастерски распознаем яблоки. Точно так же учимся различать яблоки и апельсины.

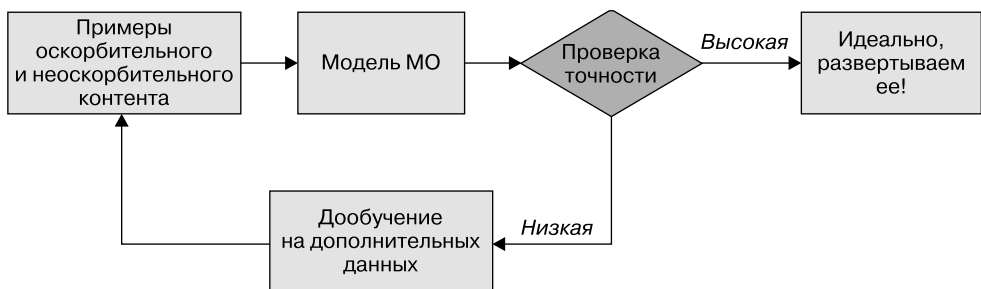


Рис. 13.1. Программирование фильтра оскорбительного контента методами МО

После обучения модель сможет сказать нам, является ли оскорбительным новое сообщение, отправленное пользователем. Она может ошибаться, как и малоопытный

ребенок, поэтому мы должны снова и снова обучать ее на дополнительных данных и оценивать точность ответов путем тестирования на немаркированных данных. Немаркированные данные, используемые для проверки точности модели, называют *тестовым набором*, а данные, применяемые для обучения, — *обучающим набором*. Как только точность модели становится достаточно высокой, ее внедряют в производство. Впоследствии она должна постоянно обучаться на новых данных, получаемых в процессе ее эксплуатации, чтобы улавливать новые слова и варианты.



Машинное обучение на маркированных данных называется обучением с учителем. Алгоритмы МО могут обучать также с использованием немаркированных данных, и в этом случае они пытаются автоматически изучить закономерности на основе представленных данных. Этот тип обучения называется обучением без учителя.

Таким образом, можно сказать, что рабочий процесс программирования МО состоит из сбора большого количества данных, соответствующей их маркировки, разделения на обучающий и тестовый наборы, использования обучающего набора для обучения модели МО, оценки эффективности модели с помощью тестового набора, развертывания и продолжения обучения. В числе популярных фреймворков машинного обучения, помогающих создавать такие модели, можно назвать *scikit-learn*, *PyTorch* и *TensorFlow*. Машинное обучение нашло применение в таких сферах, как медицина, банковское дело, социальные сети и т. д., и постоянно проникает в новые области. Далее мы коснемся аспектов тестирования.

Тестирование приложений МО

Большинство приложений МО используют типичную архитектуру на основе сервисов с компонентом МО, интегрированным в сервисы. В примере с фильтром контента процесс фильтрации на основе сервиса мог бы выглядеть так: пользователь создает новую публикацию, пользовательский интерфейс отправляет ее в сервис обработки контента, сервис проверяет ее с помощью модели и, если она определяет контент как оскорбительный, сообщает пользовательскому интерфейсу, что тот должен скрыть контент. Итак, в дополнение к обычному подходу к тестированию на основе типичной сервис-ориентированной архитектуры мы должны включить в тестирование приложения следующие аспекты.

Проверка обучающих данных

Данные, которые подаются на вход модели, во многом определяют ее качество. Если они низкокачественные, то и модель получится низкокачественной. Поэтому пристальное внимание к качеству входных данных имеет решающее значение для приложений МО. Для обучения модели нужен огромный объем данных, и их можно получить из разных источников, таких как общедоступные базы данных, общедоступные веб-сайты, разные веб-сайты (пользовательские данные) и даже системные логи. Обычно такие данные имеют самые разные

формы и размеры — практически хаотичные и неупорядоченные. В нашем примере источником данных служат публикации в социальных сетях. Помимо текста, они могут содержать изображения, видео, GIF-файлы, комментарии, теги и т. д., которые могут иметь разные размеры, форматы, цветовые градиенты и многие другие отличительные признаки. Если мы загрузим в модель такие противоречивые данные, то ей будет трудно сосредоточиться на особенностях контента, делающих его оскорбительным, таких как ключевые слова, и точно изучить различия.

Поэтому обычной практикой считается очистка входных данных, устранение шума, преобразование в стандартизированный формат и дальнейшая передача предварительно обработанных данных в модель для обучения. Логiku очистки и преобразования необходимо тщательно проверить. Вот пара базовых тестовых сценариев, которые помогут понять, как это может работать:

- когда входные данные имеют разные масштабы (например, числовые данные представляют собой значения от дробных величин меньше единицы до экспоненциально больших чисел), необходимо протестировать логику очистки данных и приведения их к единому масштабу;
- если входные данные могут содержать нулевые или пустые значения, то на этапе очистки их нужно заменить значениями по умолчанию или удалить.

В общем случае данные имеют множество особенностей, характерных для предметной области, и их необходимо явно тестировать. Например, размер сообщений в социальных сетях может ограничиваться некоторым количеством символов, и его нужно проверять при оценке качества входных данных. Обычно команды также пишут модульные тесты для логики очистки и преобразования, чтобы автоматизировать проверку некоторых из тестовых сценариев.

Проверка качества модели

Качество модели измеряется с помощью различных показателей, таких как частота ошибок, корректность, матрицы ошибок, точность и полнота. Для расчета каждого из них есть определенные методы. В своем примере мы могли бы использовать точность и полноту.

- *Точность (precision)*, как следует из названия, служит оценкой способности модели правильно предсказывать результат (отношение количества истинно положительных результатов к сумме истинно и ложноположительных результатов). Например, если модель определяет 100 сообщений как оскорбительные, из которых 99 действительно являются оскорбительными, то ее индекс точности равен 0,99.
- *Полнота (recall)* показывает, сколько действительно оскорбительных сообщений было правильно идентифицировано моделью (отношение количества истинно положительных результатов к сумме истинно положительных и ложноотрицательных результатов). Если модель правильно определила как оскорбительные 99 сообщений из общего количества 110 оскорбительных сообщений, ее индекс полноты будет равен 0,90.

Фреймворки МО, упомянутые ранее, имеют встроенные средства для расчета этих показателей, поэтому мы можем писать тесты, опирающиеся на них, и с их помощью проверять новые модели в конвейере CI, вызывая его остановку, если значения показателей окажутся ниже некоторого порога. Также имеется инструмент с открытым исходным кодом MLflow (<https://mlflow.org/>), с помощью которого можно оценить качество каждой версии модели.

Проверка предвзятости модели

Низкое качество данных — не самое худшее, гораздо хуже предвзятость модели. Не так давно резкой общественной критике подвергся алгоритм МО, используемый в Twitter для обрезки изображений (<https://oreil.ly/IVoeN>), который, выбирая лица для показа в миниатюрах, отдавал предпочтение белым людям перед чернокожими, из-за чего компании пришлось отказаться от автоматической обрезки. Подобная предвзятость модели обусловлена входными данными. Если они включают большое количество образцов, представляющих определенную демографическую группу, то модель будет отдавать предпочтение этой группе. Поэтому крайне важно проверить и входные данные, и модель МО на наличие предвзятости. В этом может помочь Facets (<https://oreil.ly/wVyQt>) — инструмент с открытым исходным кодом, позволяющий визуализировать закономерности во входных данных.

Проверка интеграции

Интеграцию между тремя уровнями — в частности, уровнями данных, модели и API — необходимо тестировать обычными методами контрактного и интеграционного тестирования.

Сосредоточившись на этих аспектах, вы сможете обеспечить непрерывную доставку. Дисциплина непрерывной доставки моделей машинного обучения (Continuous Delivery for Machine Learning, CD4ML) подробно обсуждается некоторыми моими коллегами в их статье (<https://oreil.ly/3v0gl>) на сайте Мартина Фаулера.

Блокчейн

Сэр Джон Харгрейв и Эван Карнупакис дают простое и короткое определение блокчейна в своем отчете *What Is Blockchain* (<https://oreil.ly/SNWNJ>): «Блокчейн — это *Интернет денег*». Если исходить из того, что деньги — это нечто ценное, например, акции, облигации, бонусные баллы и т. д., а Интернет — платформа для свободного обмена информацией, то блокчейн можно понимать как платформу для обмена чем-либо ценным.

Название отражает способ работы этой технологии. Всякий раз, когда совершается транзакция (передача ценности), создается блок с данными о ней, который привязывается к предыдущей транзакции. Под привязыванием я подразумеваю вычисление хеша содержимого предыдущего блока и включение его в текущий. В результате образуется цепочка блоков (рис. 13.2).

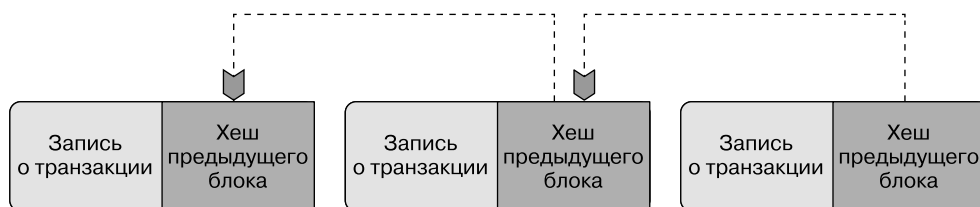


Рис. 13.2. Цепочка блоков с данными о транзакциях

Как блокчейн обеспечивает безопасность? Если кто-то изменит содержимое блока, то хеш в следующем блоке не будет соответствовать предыдущему измененному блоку и цепочка разорвется. Поэтому мы можем сказать, что транзакции неизменяемы: в цепочку можно добавлять новые блоки, но существующие изменять нельзя. Обычно для хеширования используются высокопроизводительные алгоритмы шифрования, такие как SHA-256, что делает блокчейн неязвимым для хакеров.

Какая философия стояла за созданием этой непроницаемой системы? В 2008 году некто под псевдонимом Сатоши Накамото опубликовал технический документ *Bitcoin: A Peer-to-Peer Electronic Cash System* (<https://oreil.ly/50nLa>), в котором рассказывалось о новой концепции под названием «цифровые деньги», или «электронные деньги», которые могут передаваться между сторонами без участия централизованного агента, такого как банк. Мысль была проста: люди много работают, чтобы заработать деньги, и они должны иметь над ними контроль. Деньги для людей и от людей! Восторженное сообщество разработчиков тут же приступило к работе над реализацией технического документа, который превратился в современную технологию блокчейна. Хочу обратить ваше внимание на несколько ключевых моментов: блокчейн эволюционировал, способствуя децентрализации и продвижению одноранговых транзакций. Безопасность должна была быть частью этого процесса, поскольку технология предназначена для работы с деньгами.

Введение в концепции блокчейна

Обсудим составляющие блокчейна, чтобы понять, как можно реализовать тестирование.

Децентрализованные реестры

Реестр — это хранилище, содержащее все учетные данные (входящие и исходящие транзакции). Блокчейн использует децентрализованные реестры, то есть принадлежащие не какому-то одному человеку, а всем участникам. Любая сторона, намеревающаяся совершить транзакцию, получает копию реестра. Преимущество здесь в том, что этой копии можно доверять, так как никто не может изменять записи после их добавления в реестр. Однако это влечет за собой дополнительные затраты на постоянную синхронизацию всех реестров.

Узлы

Узел — это любой компьютер или сервер, участвующий в сети блокчейна. Узлы могут принадлежать одному человеку или группе людей. Каждый узел хранит копию децентрализованного реестра. Когда происходит новая транзакция, каждый узел обновляет свою копию блокчейна. Узлы общаются друг с другом, чтобы синхронизировать реестры (рис. 13.3). Этот процесс основан на так называемой *технологии распределенного реестра* (Distributed Ledger Technology, DLT) (<https://oreil.ly/hv375>).

Консенсус

В блокчейне мы имеем децентрализованные реестры с учетными данными и узлами, обеспечивающими необходимую для их хранения инфраструктуру. Банк — это централизованный орган, который может добавлять или удалять транзакции клиентов после проверки их целостности, но в блокчейне все узлы являются равноправными участниками — так кто же может добавлять новые транзакции в цепочку? Здесь в игру вступает *консенсус*.

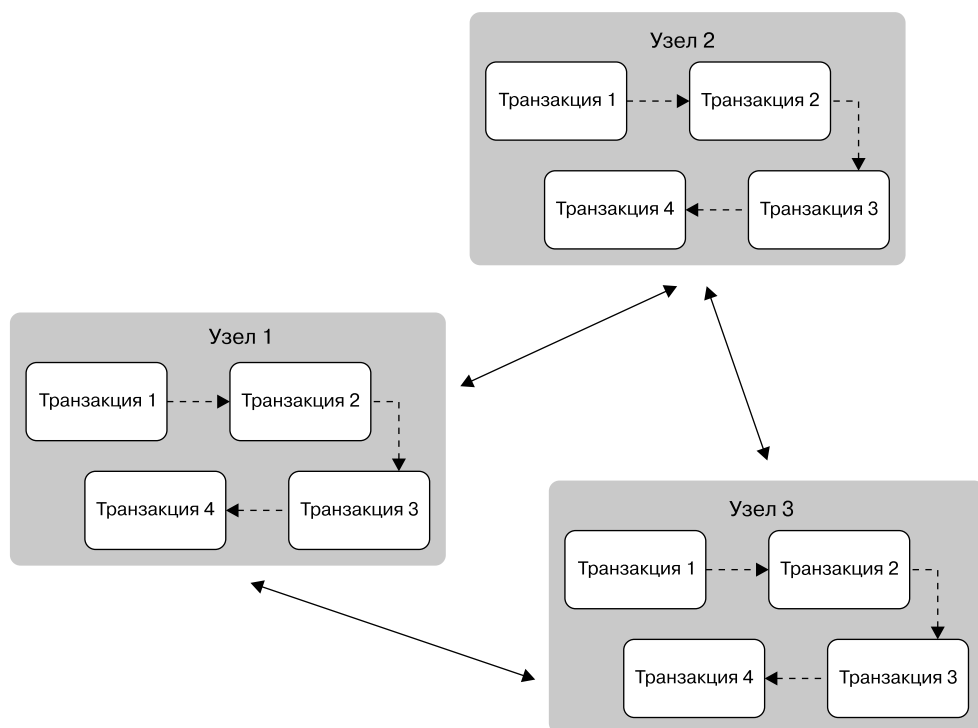


Рис. 13.3. Технология распределенного реестра с узлами, каждый из которых хранит свою копию блокчейна

Консенсус — это процесс, посредством которого узлы коллективно соглашаются добавить транзакцию. С программной точки зрения для этого используются специальные консенсусные алгоритмы, такие как *Proof of Work* («Доказательство работы») и *Proof of Stake* («Доказательство владения»). В алгоритме *Proof of Work* узлы решают чрезвычайно сложную математическую задачу. Первый узел, вычисливший правильный ответ, получает право добавить новый блок. Другие узлы проверяют целостность нового блока перед его добавлением. Когда узел добавляет новый блок, он вознаграждается цифровой валютой, это называют *майнингом* (mining). Недостаток алгоритма — необходимость использования больших вычислительных мощностей для решения сложных математических задач. В алгоритме *Proof of Stake* мощность майнинга узлов пропорциональна количеству цифровой валюты, которую они контролируют. Но и у него есть отрицательная сторона — узлы с наибольшим количеством цифровой валюты продолжают получать привилегии и богатеть.

Смарт-контракты

Банковская система имеет набор установленных правил и условий для успешного выполнения транзакции. Например, прежде, чем одобрить ипотечный кредит, банк проверяет вашу зарплату, баланс счета, документы на жилье и т. д. В блокчейне логика, необходимая для завершения транзакции, записана в виде смарт-контракта. Копию смарт-контракта получает каждый узел. Преимущества этого подхода в том, что он позволяет проводить безбумажные транзакции, устраняет комиссии для посредников и облегчает сторонам самостоятельное завершение транзакции.

Это основные составляющие блокчейна. Чтобы собрать все это воедино и получить представление об общем рабочем процессе, рассмотрим пример. Предположим, Алиса хочет купить у Боба помидоры за десять эфириумов (популярная криптовалюта). Она инициирует транзакцию и переводит деньги. Смарт-контракт удерживает деньги от Алисы до тех пор, пока Боб не доставит помидоры. В качестве доказательства доставки Боб может предоставить QR-код. Когда Алиса отсканирует его, транзакция завершится и смарт-контракт передаст деньги Бобу. Если же ему не удастся доставить помидоры, то через определенное время смарт-контракт возвращает деньги Алисе. А между тем узлы сети соревнуются за решение математической задачи и получение права добавить новый блок для этой транзакции. Победивший узел также выбирает записи транзакций из смарт-контракта для добавления в блок. После добавления блока реестр синхронизируется со всеми остальными узлами.

В числе блокчейн-фреймворков, поддерживающих все это, можно назвать Ethereum, HyperLedger Fabric и Stellar. Для написания смарт-контрактов применяются OpenZeppelin и Solidity. А в роли кошелька для цифровой валюты (в частности, эфириумов) широко используется MetaMask.

Тестирование блокчейн-приложений

Познакомившись в общих чертах с рабочим процессом в технологии блокчейна, рассмотрим несколько направлений тестирования.

Функциональное тестирование

Первый шаг при тестировании любого приложения — проверка сквозных функциональных потоков, как в примере с покупкой помидоров. Функциональная логика прописана в смарт-контрактах, поэтому нужно попробовать отыскать в ней лазейки. Тестовые сценарии для проверки смарт-контрактов также можно добавить в виде модульных тестов.

Тестирование API

Чаще всего поверх блокчейна существует API, к которому подключается пользовательский интерфейс. Мы должны сосредоточиться на стандартном тестировании уровня API: функциональности, интеграции между модулями, управлении версиями контракта, обработке ошибок, повторных попытках и т. д.

Тестирование безопасности

При тестировании безопасности необходимо проверить множество аспектов, от создания учетных записей и механизмов авторизации участников транзакций до обмена валют, поддержания баланса счета, проверки законности транзакций и криптографических аспектов, таких как хеширование блоков.

Тестирование производительности

Учитывая, что транзакции зависят от доступности узлов и алгоритмов консенсуса, для завершения транзакции может потребоваться больше времени, чем при использовании стандартной веб-технологии. Следовательно, необходимо протестировать производительность транзакций и функциональное поведение для устранения задержек.

ТЕСТИРОВАНИЕ, СПЕЦИФИЧЕСКОЕ ДЛЯ БЛОКЧЕЙНА

Большинство приложений используют существующие сети блокчейна, такие как Ethereum, для развертывания своих смарт-контрактов, поэтому вам может не потребоваться тестировать специфические функции блокчейна. Но если все же понадобится, то обратите внимание на следующие аспекты.

- *Добавление транзакций.* Любая транзакция должна регистрироваться без потери информации. Это самое важное требование в блокчейне. Блоки должны быть правильно связаны и синхронизированы со всеми остальными узлами.
- *Размер блока.* Транзакции объединяются в один блок до тех пор, пока его размер не достигнет верхнего предела (например, в сети Bitcoin предельный размер блока составляет 1 Мбайт). Мы должны проверить, создается ли новый блок, когда размер текущего блока достигает предела.

- *Размер цепочки.* С ростом количества транзакций цепочка становится очень длинной. Мы должны проверить производительность приложения при работе с цепочками большого размера.
- *Тестирование узла.* Узлы — это основа сети блокчейна. Узлы должны иметь возможность участвовать в консенсусе и постоянно синхронизировать данные. Новые узлы должны иметь возможность беспрепятственно подключаться к сети.
- *Устойчивость.* Когда узлы включаются в работу после непродолжительного простоя, они должны плавно интегрироваться в сеть, не нарушая функциональности приложения. Если в течение некоторого времени какие-то узлы остаются недоступными, приложение должно корректно обработать сбой.
- *Коллизии.* Иногда случается так, что сразу несколько узлов решили математическую задачу и борются за право добавить новую транзакцию. Мы должны протестировать сценарии коллизий.
- *Повреждение данных.* Узел, не соблюдающий правила и нормы поведения в децентрализованной системе, называют византийским (<https://oreil.ly/7Yj4j>). Взаимодействие с ним может привести к повреждению данных на других узлах. Существуют проверенные способы преодоления таких ситуаций, соответственно, такое поведение необходимо протестировать.

Для тестирования приложений блокчейна на основе Ethereum вам могут пригодиться такие инструменты, как Ethereum Tester (<https://oreil.ly/PuEWT>) и Populus (<https://oreil.ly/epHUN>), а на основе Bitcoin — bitcoinj (<https://bitcoinj.org>) и testnet (<https://oreil.ly/8zyWG>).

Как видите, технология блокчейна имеет существенные преимущества с точки зрения безопасности, полностью цифровых транзакций, устранения посредников и борьбы с монополией. Однако есть и некоторые недостатки, затрудняющие внедрение этой технологии. Например, блокчейн требует большой вычислительной мощности и много электроэнергии для решения сложных математических задач и синхронизации данных в реестре, а из-за алгоритмов консенсуса и периодической недоступности узлов завершение транзакции может занять много времени. Сообщается, что Visa обрабатывает около 1700 транзакций в секунду (<https://oreil.ly/f7hLq>), тогда как подтверждение одной транзакции блокчейна может занять 10 мин. Соответственно, производительность — ее основное узкое место.

Интернет вещей

Интернет вещей (Internet of Things, IoT) — это технология, связывающая физический мир с цифровым. Она позволяет нашим устройствам (вещам) получать информацию и общаться друг с другом и с нами через Интернет. Также эта технология позволяет устройствам реагировать на изменения в окружающей среде без вмешательства человека. Например, интеллектуальные термостаты оценивают атмосферные условия, такие как влажность, и устанавливают правильную температуру в зависимости от предпочтений пользователя. Интернет

вещей хорошо зарекомендовал себя как средство удовлетворения и небольших, и крупномасштабных потребностей. Известным примером является решение «умный дом». Ожидается, что в 2022 году глобальная стоимость рынка «умных домов» превысит 53 млрд долларов (<https://oreil.ly/01Bct>). На другом конце спектра находятся решения IoT для «умных городов» (<https://oreil.ly/Yap9V>), которые стремятся повысить общее качество жизни горожан за счет улучшения инфраструктуры, качества воздуха, доступности транспорта, потребления энергии и многого другого.

Устройства Интернета вещей обычно оснащены тремя функциями: датчиком, исполнительным механизмом и средствами связи. Датчики определяют физические условия, такие как температура, частота пульса, скорость движения и т. д. Исполнительные механизмы вызывают изменения в окружающей среде, например подают сигнал тревоги при обнаружении дыма или открывают и закрывают клапаны для управления температурой. Средства связи, такие как цифровые дисплеи и голосовое управление, помогают устройствам Интернета вещей предоставлять информацию пользователю.

Создание комплексного решения Интернета вещей требует навыков работы как с аппаратным, так и с программным обеспечением. Программный компонент встроен в аппаратное обеспечение для управления его функциями и передачи информации пользователям. Другой программный компонент находится вне аппаратного обеспечения, он собирает и анализирует данные, отправленные с нескольких устройств, для выполнения коллективных действий. Например, чтобы получить частоту пульса пользователя с помощью фитнес-устройства, программное обеспечение внутри него включает аппаратный датчик измерения пульса и передает результат на цифровой дисплей. Также ПО отправляет эту информацию в облако, где сервис анализирует информацию о частоте пульса, циклах сна и так далее и дает указание встроенному программному обеспечению подать сигнал тревоги при обнаружении аномалий.

Чтобы все эти технологии — датчики, сети, протоколы связи и маршрутизации, процессоры данных, приложения конечных пользователей, облако и многое другое — работали вместе, требуется комплексная интеграция. Давайте поближе рассмотрим пятиуровневую архитектуру Интернета вещей — это поможет вам лучше понять интеграцию.

Введение в пятиуровневую архитектуру Интернета вещей

Существуют разные точки зрения на определение количества уровней в архитектуре Интернета вещей — три, четыре или пять (<https://oreil.ly/TNAJq>). Пятиуровневая архитектура (рис. 13.4) дает более широкое и глубокое представление о технологиях, задействованных в создании фулстек-приложения IoT. Мы кратко рассмотрим каждый из этих уровней, чтобы понять аспекты их тестирования.



Рис. 13.4. Пятиуровневая архитектура Интернета вещей

Кратко рассмотрим каждый из этих уровней, чтобы понять, какие аспекты тестирования связаны с каждым из них.

Уровень восприятия

Это самый нижний уровень, на котором оборудование получает информацию из физического мира и передает ее на следующие уровни. Аппаратное обеспечение на этом уровне можно разделить на пассивное, полупассивное и активное в зависимости от поддержки однонаправленной или двунаправленной связи. Например, сканеры QR-кодов — это пассивные устройства, потому что они могут общаться только в одном направлении, а диапазон связи ограничен. Этого достаточно для таких сценариев, как отслеживание доставки. Также обратите внимание на то, что пассивные компоненты недостаточно мощны для выполнения вычислений. Активные компоненты могут принимать и передавать данные и наделены необходимыми для этого возможностями. Примерами могут служить интеллектуальные приводы, выполняющие механические задачи, носимые устройства со встроенными датчиками, радиоприемники GPS и т. д. Они также могут общаться на больших расстояниях.

Сетевой уровень

Физические устройства должны идентифицироваться в Интернете, чтобы другие устройства могли с ними взаимодействовать. IPv4 и IPv6 — популярные сетевые протоколы, которые предоставляют устройствам уникальные IP-адреса (предпочтительнее IPv6). Для эффективной передачи данных в устройства

задействуются протоколы маршрутизации, такие как протокол маршрутизации для сетей с низким энергопотреблением и потерями (Routing Protocol for Low-Power and Lossy Networks, RPL). Для передачи и получения информации используются стандартные коммуникационные технологии, такие как Wi-Fi, Zigbee, NFC и Bluetooth.

Промежуточный уровень

Приложения IoT должны иметь возможность обращаться к физическим устройствам по их именам или адресам (например, чтобы получить температуру воздуха в помещении, частоту пульса пользователя и т. д.), не вникая в детали базовой инфраструктуры. Промежуточный уровень помогает обнаружить такие сервисы. Он также занимается извлечением данных из физических устройств и передачей информации обратно пользователям. Это ядро решения Интернета вещей. В нем широко задействуются протоколы обнаружения служб, такие как Avahi и Bonjour, а также обмена данными, такие как Constrained Application Protocol (CoAP) и Message Queuing Telemetry Transport (MQTT).

Уровень приложения

Этот уровень позволяет конечным пользователям получать доступ к нужным сервисам через простой интерфейс в виде мобильного или веб-приложения, и он не заботится о том, как обрабатываются запросы на нижележащих уровнях. Уровень приложения включает в себя логику сбора, обработки и хранения информации с нескольких устройств.

Бизнес-уровень

Этот уровень анализирует информацию, полученную от оборудования, сервисов и т. д., для предоставления усовершенствованных услуг. Для анализа огромных объемов данных, полученных от различных устройств Интернета вещей, на этом уровне используются технологии больших данных (big data), такие как Apache Spark и Apache Kafka. Этот уровень в основном предназначен для внутренних административных, а не для конечных пользователей.

Платформы Интернета вещей, такие как AWS IoT и IBM Watson, сочетают в себе многие из этих возможностей, упрощая разработку приложений для данной сферы.

Тестирование приложений Интернета вещей

Далее перечислены некоторые особенности, на которые следует обратить внимание при тестировании решений Интернета вещей.

Интеграция программного/аппаратного обеспечения

Сквозная функциональность любого приложения для Интернета вещей во многом зависит от правильной интеграции аппаратного и программного обеспечения, и, тестируя ее, необходимо проверить всевозможные крайние случаи.

Например, приложение для мониторинга сердцебиения в «умных» часах должно отображать правильное количество сердцебиений, записанное датчиком, а при возникновении проблем с записью сердцебиения программное обеспечение должно корректно обрабатывать ошибки. Тестирование интеграции следует проводить после установки или обновления ПО и оборудования. Кроме того, при тестировании функциональных возможностей необходимо учитывать обычные аппаратные ограничения с точки зрения памяти и батареи.

Сеть

Сетевое соединение устройства с облаком — важный аспект решений Интернета вещей, который необходимо тщательно протестировать. Некоторые устройства могут поддерживать несколько протоколов связи, например Wi-Fi и Bluetooth, и эти возможности необходимо тестировать независимо.

Функциональная совместимость

Функциональная совместимость в решениях Интернета вещей означает способность различных устройств обмениваться информацией друг с другом, даже если они реализуют разные стандарты и протоколы. Например, в интеллектуальном решении организации дорожного движения устройства датчиков, служб обнаружения аварий и систем автоматической маршрутизации должны иметь возможность беспрепятственно обмениваться информацией, даже если каждое из них использует разные наборы технологий и протоколов. Функциональная совместимость по-настоящему раскрывает потенциал Интернета вещей, поэтому интеграцию необходимо тщательно протестировать.

Безопасность и конфиденциальность

Некоторые протоколы связи, такие как Z-Wave, не всегда могут обеспечить должный уровень безопасности, поэтому для предотвращения атак необходимо использовать дополнительные облегченные механизмы безопасности, такие как IPsec. Кроме того, должна обеспечиваться конфиденциальность данных, хранящихся в облаке. Хранить биометрические и другие личные данные людей без их согласия неэтично, к тому же, как мы видели в главе 10, существуют законодательные требования к хранению личной информации, и мы должны проверять соответствие им.

Производительность

Производительность — важный аспект качества решений Интернета вещей, поскольку может существовать множество устройств, взаимодействующих друг с другом и передающих информацию обратно сервисам агрегатора. Соответственно, мы должны проверить, как быстро оборудование реагирует на команды программного обеспечения, каково общее время ответа сервиса (например, время получения частоты пульса) и быстро ли происходит сбор данных, когда в сети действует много устройств, например, как в «умном городе».

Удобство использования

Удобство использования — критически важный аспект, особенно в бытовом секторе, например, для таких устройств, как «умные» часы и смарт-телевизоры. Для них может потребоваться протестировать множество самых разных аспектов. «Умные» часы реагируют на движения запястья, имеют дисплеи разных размеров, могут управляться разными кнопками и жестами, имеют систему оповещения в виде вибраций и звуковых сигналов, их можно носить на правой или левой руке и т. д. Важной частью продукта является также знакомство пользователей с возможностями устройства. Тестирование совокупности аспектов удобства применения имеет решающее значение для успеха продукта.

Из собственного опыта работы над созданием «умной» кофемашины могу засвидетельствовать, что тестирование IoT-решений невероятно сложная задача из-за разнообразных комбинаций устройств и их внутренних состояний. Чтобы справиться с разнообразием состояний и комбинаций устройств, а также созданием тестовых сценариев, я сформулировала основу для тестирования под названием *IoT Testing Atlas* (<https://oreil.ly/uMEX2>), познакомиться с которой, я надеюсь, вам будет интересно!

Дополненная и виртуальная реальность

Дополненная реальность (ДР) — это технология наложения графики, текста, изображений и другой сенсорной информации на реальную среду для улучшения ее восприятия пользователем. Первоначально она была изобретена в помощь пилотам реактивных истребителей, которым необходимо было сконцентрировать свое внимание на атакуемой цели, и отображала информацию о состоянии систем самолета на фронтальных дисплеях, помогая им сосредоточиться на обеих задачах одновременно. Одним из последних примеров ДР является разработанный в Mercedes проекционный дисплей, отображающий такую информацию, как карты и ограничение скорости, на лобовом стекле автомобиля.

Сейчас существуют игры, в которых задействуют «умные» очки, проекционные дисплеи и различные мобильные и портативные устройства, чтобы дать игроку возможность получить опыт общения с ДР. Возможно, вы слышали о носимых «умных» дисплеях от таких производителей, как Google, Vuzix, Epson и Nreal, или пробовали их. Однако наиболее близки нам смартфоны с поддержкой ДР. Обе ведущие операционные системы для мобильных устройств, Android и iOS, оснащены инструментами и фреймворками, такими как ARCore, ARKit и Unity AR Foundation, необходимыми для реализации этой технологии, и существует множество телефонов с поддержкой дополненной реальности, например Pixel 5, Nokia 8, Moto G и т. д.

В отличие от ДР, которая дополняет реальное окружение, *виртуальная реальность* (ВР) переносит пользователя в смоделированный виртуальный мир. Помимо популярных приложений, таких как игры, эта технология используется для моделирования разного рода опасностей, например пожаров или аварий, для обучения борьбе с ними. Кроме того, ВР приобрела большую популярность в коммерческом пространстве, где клиентам предлагают такие возможности, как проектирование интерьера их нового дома или виртуальное взаимодействие с товарами, например, в виртуальных примерочных.

Для полного погружения в виртуальную реальность требуются устройства с видеощлемом. В числе популярных устройств, имеющих на рынке, можно назвать Oculus Quest, Oculus Go, HTC VIVE и Sony PlayStation VR. Более доступный и экономичный вариант — смартфоны с такими решениями, как Google Cardboard.

Помимо ДР и ВР, есть также *смешанная реальность* (СР), сочетающая в себе черты ДР и ВР и позволяющая пользователям взаимодействовать с цифровым контентом в трехмерном окружении. В качестве примера СР можно привести игру Pokemon Go (<https://oreil.ly/0IuTI>). Аналогично *расширенная реальность* (РР) обеспечивает интеграцию устройств ДР, ВР и СР с другими устройствами, такими как бытовая техника, датчики и т. д. Пространство ДР, ВР, СР и РР расширяется, и на него определенно стоит обратить внимание.

Тестирование приложений ДР/ВР

Технологии ДР и ВР увлекательны и дарят пользователям захватывающие впечатления, но при этом невероятно сложны. Разработка и тестирование этих продуктов требует знаний в широком спектре областей, от биологии (восприятие изображений человеком, механика формирования изображений глазом, восприятие глубины и т. д.) до пространственной математики, технологий отображения в видеошлеме и многого другого. Существуют платформы разработки, такие как Unity, которые до некоторой степени устраняют эти сложности. Кроме того, за прошедшие годы заметно улучшились качество и производительность шлемов виртуальной реальности.

Однако, несмотря на рост популярности, в этой области все еще недостаточно и инструментов, и устоявшихся методик тестирования. Всякий раз тестирование осуществляется контекстно. Не так давно в Thoughtworks был создан инструмент автоматизации функционального тестирования для Unity под названием Arium (<https://oreil.ly/0F6mV>). Arium имеет открытый исходный код и доступен в виде пакета Unity. Давайте кратко рассмотрим несколько концепций Unity, чтобы понять, как тестировать с помощью этого инструмента¹.

¹ Более полное введение вы найдете в книге Кейси Хардмана *Game Programming with Unity and C#: A Complete Beginner's Guide* (Apress).

Сцена в Unity представляет игровую среду. Обычно каждый уровень в игре называется сценой. Каждая сцена имеет свою коллекцию объектов. *Игровой объект* — это элемент сцены. Это может быть предмет, допустим мяч, или игровой персонаж. Способности этих объектов можно программировать, прикрепляя к ним *компоненты* (компонент — это любая функция или особенность, связанная с игровым объектом). Unity предоставляет множество встроенных компонентов, охватывающих основные потребности, таких как освещение, коллизии и т. д. Например, мы можем прикрепить компонент освещения к игровому объекту, чтобы определить, как он должен освещаться. Каждый игровой объект имеет также компонент *трансформации* по умолчанию, который представляет его положение, размер и поворот.

Agium предлагает следующие возможности для автоматизации функционального тестирования приложений Unity:

- `_arium.FindGameObject("Ball")` для поиска игрового объекта по имени;
- `_arium.GetComponent<имя_компонента>(<имя_игрового_объекта>)` для получения компонента из указанного игрового объекта, который затем можно будет проверить;
- `_arium.PerformAction(new UnityPointerClick(), "<имя_игрового_объекта>")` для выполнения действий над игровыми объектами.

Agium можно использовать для тестирования удобства применения, экспериментального тестирования и тестирования эффекта присутствия, производительности и совместимости приложений смешанной реальности.

На этом я заканчиваю краткое описание новых технологий и некоторых аспектов их тестирования, о которых следует помнить. Эти технологии продолжают развиваться и могут занять лидирующие позиции раньше, чем мы предполагаем, поэтому давайте следить за этой областью!

Об авторе

Гаятри Мохан — увлеченный техлид и эксперт в различных областях разработки программного обеспечения и промышленных технологий. Гаятри не раз доказывала свою компетентность, успешно управляя большими командами обеспечения качества в Thoughtworks, где в настоящее время занимает должность главного консультанта. Будучи директором технического обеспечения, Гаятри поддерживала местные технологические сообщества, организовывала технические мероприятия и развивала интеллектуальное лидерство по техническим темам.

Гаятри является соавтором книги *Perspectives of Agile Software Testing* (<https://oreil.ly/PoAST>), выпущенной компанией Thoughtworks к десятилетию Selenium.

Иллюстрация на обложке

На обложке изображен низинный полосатый тенрек (*Hemicentetes semispinosus*). Эти мелкие насекомоядные млекопитающие — один из многих видов тенреков, обитающих на острове Мадагаскар. Полосатые тенреки обычно живут в кустарниках низинных тропических лесов, в сельскохозяйственных угодьях и даже в сельских садах в восточной части острова.

Низинных полосатых тенреков легко узнать по длинной заостренной черной морде и маленькому бесхвостому телу с полосчатой желто-черной окраской игл. Тылльную сторону шеи тенрека прикрывает гребень из желтых шипов. Иглы животного могут служить средством обороны. Тенреки используют их также для общения — при трении друг о друга иглы издают высокий пронзительный звук. Длина игл взрослого полосатого тенрека может достигать 18 см, а его вес — 280 г.

Низинные полосатые тенреки общительны и собираются в группы до 20 особей. Они роют сообщающиеся норы для гнездования и добычи дождевых червей и насекомых совместно или поодиночке. Зимой они впадают в оцепенение — температура их тела понижается и обмен веществ замедляется. Самки могут приносить потомство только в течение первого года жизни и достигают репродуктивного возраста уже через 25 дней после рождения, что делает их единственным видом тенреков, которые могут размножаться в том же сезоне, в котором появились на свет. Низинные полосатые тенреки классифицируются Международным союзом охраны природы как вид, вызывающий наименьшее беспокойство, из-за их широкого распространения, большой численности и способности жить рядом с людьми.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой вымирания, все они очень важны для биосферы.

Иллюстрацию для обложки нарисовала Карен Монтгомери, взяв за основу черно-белую гравюру из книги *English Cyclopedia*.

Гаятри Мохан

Фулстек-тестирование

Перевел с английского А. Киселев

Научные редакторы А. Захарова, Д. Колфилд

Изготовлено в Казахстане. Изготовитель: ТОО «Спринт Бук».

Место нахождения и фактический адрес: 010000 Казахстан, город Астана, район Алматы,
Проспект Рахымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 06.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 07.06.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 33,540. Заказ 0000.

Отпечатано в ТОО «ФАРОС Графика». 100004, РК, г. Караганда, ул. Молокова, 106/2.



Оливье Келен, Мари-Алис Блете

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА БАЗЕ GPT-4 И CHATGPT

Эта небольшая книга представляет собой подробное руководство для разработчиков на Python, желающих научиться создавать приложения с использованием больших языковых моделей. Авторы расскажут об основных возможностях и преимуществах GPT-4 и ChatGPT, а также о принципах их работы. Здесь же вы найдете пошаговые инструкции по разработке приложений с использованием библиотеки поддержки GPT-4 и ChatGPT для Python, в том числе инструментов для генерирования текста, отправки вопросов и получения ответов и обобщения контента.

«Разработка приложений на базе GPT-4 и ChatGPT» содержит множество легковоспроизводимых примеров, которые помогут освоить особенности применения моделей в своих проектах. Все примеры кода на Python доступны в репозитории GitHub. Решили использовать возможности LLM в своих приложениях? Тогда вы выбрали правильную книгу.

Кент Бек

ЧИСТЫЙ ДИЗАЙН. ПРАКТИКА ЭМПИРИЧЕСКОГО ПРОЕКТИРОВАНИЯ ПО



Грязный код создает проблемы. Чтобы код было проще читать, приходится проводить его очистку, разбивая на части, с которыми удобно работать. Кент Бек, создатель методологии экстремального программирования и первопроходец в области паттернов проектирования, рассказывает нам, где и когда лучше проводить очистку для улучшения кода с учетом общей структуры системы.

Книга не заставляет читателя проводить очистку сразу и целиком, а позволяет протестировать несколько примеров, которые подходят для поставленной задачи. Вы узнаете, как логически разделить на части большую функцию, содержащую множество строк кода. Познакомитесь с теоретическими понятиями программного дизайна: сцеплением, связностью, дисконтированными денежными потоками и вариативностью.

КРОК

СОЗДАЕМ НАСТОЯЩЕЕ,
ИНТЕГРИРУЕМ БУДУЩЕЕ



croc.ru

КРОК — технологический партнер с комплексной экспертизой в области построения и развития инфраструктуры, внедрения информационных систем, разработки программных решений и сервисной поддержки.

Центры компетенций КРОК фокусируются на ключевых отраслевых кластерах — промышленность, финансовый сектор, розничные продажи, муниципальное управление, спорт и культура.

Ежегодно сотни проектов КРОК становятся системообразующими для экономики и социально-культурной сферы.

