



ПРАКТИКА НЕПРЕРЫВНЫХ АПДЕЙТОВ

CONTINUOUS DELIVERY



A Practical Guide to Continuous Delivery

Eberhard Wolff

◆ Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco
Amsterdam • Cape Town • Dubai • London • Madrid • Milan
Munich • Paris • Montreal • Toronto • Delhi • Mexico City • São Paulo
Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Э Б Е Р Х А Р Д В О Л Ь Ф

ПРАКТИКА НЕПРЕРЫВНЫХ АПДЕЙТОВ

CONTINUOUS DELIVERY



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2018

ББК 32.973.2-018
УДК 004.45
В72

Вольф Эберхард

В72 Continuous delivery. Практика непрерывных апдейтов. — СПб.: Питер, 2018. — 320 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0480-2

Эта книга поможет всем, кто собирается перейти на непрерывную поставку программного обеспечения. Руководители проектов ознакомятся с основными процессами, преимуществами и техническими требованиями. Разработчики, администраторы и архитекторы получают необходимые навыки организации работы, а также узнают, как непрерывная поставка внедряется в архитектуру программного обеспечения и структуру ИТ-организации.

Эберхард Вольф познакомит вас с популярными передовыми технологиями, облегчающими труд разработчиков: Docker, Chef, Vagrant, Jenkins, Graphite, ELK stack, JBehave и Gatling. Вы пройдете через все этапы сборки, непрерывной интеграции, нагрузочного тестирования, развёртывания и контроля.

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

ББК 32.973.2-018
УДК 004.45

Права на издание получены по соглашению с Addison-Wesley Longman. 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.

ISBN 978-0134691473 англ.
ISBN 978-5-4461-0480-2

© 2017 Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2018
© Издание на русском языке, оформление ООО Издательство «Питер», 2018
© Серия «Для профессионалов», 2018

Краткое содержание

Часть I. Основы.....	31
Глава 1. Непрерывное развертывание: что и как?	32
Глава 2. Подготовка инфраструктуры	52
Часть II. Конвейер непрерывного развертывания	113
Глава 3. Автоматизация сборки и непрерывная интеграция	114
Глава 4. Приемочные тесты	161
Глава 5. Тестирование пропускной способности	192
Глава 6. Исследовательское тестирование	210
Глава 7. Развертывание — ввод в эксплуатацию.....	219
Глава 8. Эксплуатация	234
Часть III. Управление, организация и архитектура решения непрерывного развертывания.....	267
Глава 9. Внедрение методологии непрерывного развертывания на предприятии.....	268
Глава 10. Непрерывное развертывание и DevOps	279
Глава 11. Непрерывное развертывание, DevOps и архитектура ПО	293
Глава 12. Заключение: основные преимущества.....	315

Оглавление

Предисловие	18
П.1. Обзор методологии непрерывного развертывания и книги	18
П.2. Зачем нужно непрерывное развертывание?	19
П.2.1. Короткая история	19
П.2.2. Непрерывное развертывание способно помочь в таких ситуациях.....	21
П.3. Кому адресована эта книга?	23
П.4. Краткое содержание глав.....	23
П.5. Как читать эту книгу	25
Благодарности	28
Об авторе.....	29
От издательства	30
Часть I. Основы.....	31
Глава 1. Непрерывное развертывание: что и как?.....	32
1.1. Введение: что такое непрерывное развертывание?	32
1.2. Почему процесс выпуска ПО настолько сложен	32
1.2.1. Непрерывная интеграция вселяет надежду.....	33
1.2.2. Медленные и опасные процессы	33
1.2.3. Есть возможность ускорения.....	33
1.3. Ценность непрерывного развертывания	34
1.3.1. Регулярность.....	35
1.3.2. Контролируемость.....	35
1.3.3. Регрессии.....	36

1.4. Преимущества непрерывного развертывания	37
1.4.1. Непрерывное развертывание сокращает сроки внедрения	37
1.4.2. Один пример	37
1.4.3. Реализация услуги и ее внедрение в эксплуатацию.....	38
1.4.4. Следующий шаг	38
1.4.5. Непрерывное развертывание дает конкурентные преимущества	38
1.4.6. Без непрерывного развертывания	38
1.4.7. Непрерывное развертывание и Lean Startup	39
1.4.8. Влияние непрерывного развертывания	40
1.4.9. Непрерывное развертывание для минимизации рисков	41
1.4.10. Быстрая обратная связь и низкие затраты.....	45
1.5. Создание и организация конвейера непрерывного развертывания	46
1.5.1. Пример	49
1.6. В заключение	50
Ссылки	50
Глава 2. Подготовка инфраструктуры	52
2.1. Введение	52
2.1.1. Автоматизация инфраструктуры: пример	53
2.2. Сценарии установки	54
2.2.1. Проблемы классических сценариев установки.....	54
2.3. Chef	57
2.3.1. Chef и Puppet	59
2.3.2. Другие альтернативы	60
2.3.3. Технические основы.....	61
Основные термины Chef	62
Повара, поваренные книги и рецепты.....	63
Роли	66
2.3.4. Chef Solo	68
2.3.5. Chef Solo: заключение.....	70
2.3.6. Knife и Chef Server	70
2.3.7. Chef Server: заключение.....	75

2.4. Vagrant	76
2.4.1. Пример с Chef и Vagrant	78
2.4.2. Vagrant: заключение	80
2.5. Docker	80
2.5.1. Решение на основе Docker	81
Контейнеры Docker и виртуализация	83
Цели Docker	83
Взаимодействия между контейнерами Docker	83
2.5.2. Создание контейнеров Docker	84
Файлы Dockerfile	85
Сборка и запуск образов Docker	86
2.5.3. Запуск примера приложения с помощью Docker	87
Дополнительные команды Docker	88
2.5.4. Docker и Vagrant	89
Комплектация контейнеров с помощью Vagrant	89
2.5.5. Docker Machine	92
2.5.6. Сложные конфигурации Docker	94
Docker Registry	94
Docker в кластере	95
2.5.7. Docker Compose	96
2.6. Неизменяемый сервер	99
2.6.1. Недостатки идемпотентности	99
2.6.2. Неизменяемый сервер и Docker	100
2.7. Инфраструктура как код	101
2.7.1. Тестирование инфраструктуры как кода	103
Serverspec	103
Test Kitchen	103
ChefSpec	103
2.8. Платформа как услуга	104
2.9. Хранение информации и базы данных	106
2.9.1. Обработка схем	107
2.9.2. Тестовые и базовые данные	108
2.10. В заключение	109
Ссылки	110

Часть II. Конвейер непрерывного развертывания	113
Глава 3. Автоматизация сборки и непрерывная интеграция	114
3.1. Введение	114
3.1.1. Автоматизация сборки: пример	115
3.2. Автоматизация сборки и инструменты сборки	115
3.2.1. Инструменты сборки в мире Java	116
3.2.2. Ant	117
3.2.3. Maven	118
Версии и мгновенные снимки	121
Выпуск версий с помощью Maven	122
3.2.4. Gradle	123
Gradle Wrapper	125
3.2.5. Другие инструменты сборки	126
3.2.6. Выбор правильного инструмента	127
3.3. Модульные тесты	129
3.3.1. Создание хороших модульных тестов	130
3.3.2. Разработка через тестирование	133
3.3.3. Чистый код и искусство программирования	134
3.4. Непрерывная интеграция	135
3.4.1. Jenkins	136
Расширение с помощью плагинов	138
Плагин SCM Sync Configuration	138
Плагин Environment Injector	138
Плагин Job Configuration History	139
Плагин Clone Workspace SCM	139
Плагин Build Pipeline	139
Плагин Amazon EC2	140
Плагин Job DSL	140
Создание собственных плагинов	142
3.4.2. Инфраструктура непрерывной интеграции	142
3.4.3. В заключение	144
3.5. Оценка качества кода	147
3.5.1. SonarQube	149
Интеграция в конвейер	149

3.6. Управление артефактами	152
3.6.1. Интеграция в процедуру сборки	154
3.6.2. Дополнительные особенности репозитория	156
3.7. В заключение	157
Ссылки	158
Глава 4. Приемочные тесты	161
4.1. Введение	161
4.1.1. Приемочные испытания: пример	161
4.2. Пирамида тестирования	162
4.3. Что такое «приемочные тесты»?	166
4.3.1. Автоматизированные приемочные испытания	166
4.3.2. Больше, чем просто увеличение эффективности	167
4.3.3. Тестирование вручную	168
4.3.4. Какую роль играет заказчик?	168
4.3.5. Приемочные и модульные тесты	169
4.3.6. Окружения для тестирования	170
4.4. Приемочные тесты через ГИП	171
4.4.1. Проблемы тестирования графического интерфейса	171
4.4.2. Абстракции против хрупких тестов через графический интерфейс	172
4.4.3. Автоматизация с помощью Selenium	173
4.4.4. Программный интерфейс веб-драйвера	173
4.4.5. Тестирование без веб-браузера: HtmlUnit	173
4.4.6. Программный интерфейс веб-драйвера Selenium	174
4.4.7. Selenium IDE	174
4.4.8. Проблемы автоматизации тестов через графический интерфейс	176
4.4.9. Выполнение тестов графического интерфейса	176
4.4.10. Преобразование тестов в программный код	176
4.4.11. Изменение тестов вручную	177
4.4.12. Тестовые данные	177
4.4.13. Шаблон Page Object	178

4.5. Альтернативные инструменты тестирования графического интерфейса	179
4.5.1. PhantomJS	179
4.5.2. Windmill.....	179
4.6. Текстовые приемочные тесты	181
4.6.1. Behavior-Driven Development.....	181
4.6.2. Адаптеры	184
4.7. Альтернативные фреймворки.....	186
4.8. Стратегии приемочных испытаний	187
4.8.1. Выбор инструмента	187
4.8.2. Быстрая обратная связь	188
4.8.3. Охват тестами	189
4.9. В заключение	189
Ссылки	191
Глава 5. Тестирование пропускной способности	192
5.1. Введение	192
5.1.1. Тестирование пропускной способности: пример.....	192
5.2. Тестирование пропускной способности — как?	193
5.2.1. Цели тестирования пропускной способности.....	193
5.2.2. Данные и окружения.....	194
5.2.3. Тестирование производительности должно выполняться только в конце реализации?	194
5.2.4. Тесты пропускной способности = управление рисками	194
5.2.5. Имитация поведения пользователей	195
5.2.6. Документирование требований к производительности	195
5.2.7. Аппаратное обеспечение для тестирования пропускной способности.....	196
5.2.8. Облачные решения и виртуализация	197
5.2.9. Минимизация рисков за счет непрерывного тестирования	198
5.2.10. Тестирование пропускной способности — целесообразно ли?	198
5.3. Реализация тестов пропускной способности	199
5.4. Тестирование пропускной способности с помощью Gatling.....	201
5.4.1. Различия между демонстрацией и практическим применением	205

5.5. Альтернативы инструменту Gatling	207
5.5.1. Grinder	207
5.5.2. Apache JMeter	207
5.5.3. Tsung	208
5.5.4. Коммерческие решения	208
5.6. В заключение	209
Ссылки	209
Глава 6. Исследовательское тестирование	210
6.1. Введение	210
6.1.1. Исследовательское тестирование: пример	210
6.2. Цель исследовательского тестирования	210
6.2.1. Иногда ручное тестирование оказывается предпочтительнее ...	211
6.2.2. Тестирование заказчиком	211
6.2.3. Ручное тестирование нефункциональных требований	212
6.3. Как это сделать?	212
6.3.1. Руководство по проведению тестирования	213
6.3.2. Автоматизированное окружение	213
6.3.3. Демонстрационные примеры как основа	213
6.3.4. Пример: приложение электронной коммерции	214
6.3.5. Бета-тестирование	214
6.3.6. Сеансовые тесты	215
6.4. В заключение	218
Ссылки	218
Глава 7. Развертывание — ввод в эксплуатацию	219
7.1. Введение	219
7.1.1. Развертывание: пример	220
7.2. Ввод в эксплуатацию и откат	220
7.2.1. Преимущества	220
7.2.2. Недостатки	221
7.3. Развертывание исправлений	222
7.3.1. Преимущества	222
7.3.2. Недостатки	222

7.4. Сине-зеленое развертывание.....	223
7.4.1. Преимущества.....	224
7.4.2. Недостатки.....	224
7.5. Канареечное развертывание.....	225
7.5.1. Преимущества.....	226
7.5.2. Недостатки.....	226
7.6. Автоматическое развертывание.....	227
7.6.1. Преимущества.....	228
7.6.2. Недостатки.....	229
7.7. Виртуализация.....	229
7.7.1. Физические хосты.....	231
7.8. Вне круга веб-приложений.....	231
7.9. В заключение.....	233
Ссылки.....	233
Глава 8. Эксплуатация.....	234
8.1. Введение.....	234
8.1.1. Эксплуатация — пример.....	235
8.2. Проблемы в период эксплуатации.....	235
8.3. Файлы журналов.....	237
8.3.1. Что следует журналировать?.....	238
8.3.2. Инструменты для обработки файлов журналов.....	240
8.3.3. Журналирование в примере приложения.....	241
8.4. Анализ журналов примера приложения.....	242
8.4.1. Анализ с применением Kibana.....	245
8.4.2. ELK — масштабируемость.....	246
8.5. Другие технологии обработки журналов.....	251
8.6. Продвинутое технологии обработки журналов.....	252
8.6.1. Анонимизация.....	253
8.6.2. Производительность.....	253
8.6.3. Время.....	253
8.6.4. Эксплуатационная база данных.....	254
8.7. Мониторинг.....	254

8.8. Отображение числовых характеристик с помощью Graphite.....	255
8.9. Характеристики, измеряемые в примере приложения	258
8.9.1. Структура примера.....	258
8.10. Другие решения для мониторинга.....	260
8.11. Прочие проблемы, возникающие во время эксплуатации приложений	262
8.11.1. Сценарии	262
8.11.2. Приложения в вычислительном центре клиента.....	262
8.12. В заключение	263
Ссылки	264

Часть III. Управление, организация и архитектура решения непрерывного развертывания..... 267

Глава 9. Внедрение методологии непрерывного развертывания на предприятии	268
9.1. Введение	268
9.2. Непрерывное развертывание с самого начала	268
9.3. Систематизация потока ценностей.....	269
9.3.1. Систематизация потока ценностей описывает последовательность событий.....	270
9.3.2. Оптимизация.....	271
9.4. Дополнительные меры для оптимизации	273
9.4.1. Капиталовложения в качество	273
9.4.2. Затраты.....	273
9.4.3. Выгоды	274
9.4.4. Запрещайте сохранение изменений в случае ошибки сборки! ...	274
9.4.5. Останови конвейер	275
9.4.6. Пять почему	275
9.4.7. DevOps.....	276
9.5. В заключение	278
Ссылки	278

Глава 10. Непрерывное развертывание и DevOps 279 |

10.1. Введение	279
----------------------	-----

10.2. Что такое DevOps?	279
10.2.1. Проблемы	280
10.2.2. Точка зрения клиента	281
10.2.3. Первопроходцы: Amazon	281
10.2.4. DevOps	282
10.3. Непрерывное развертывание и DevOps	283
10.3.1. DevOps: не только непрерывное развертывание	284
10.3.2. Индивидуальная ответственность и самоорганизация	285
10.3.3. Технологические решения	286
10.3.4. Меньше централизованного управления	286
10.3.5. Плюрализм технологий	287
10.3.6. Обмен специалистами между командами	287
10.3.7. Архитектура	288
10.4. Непрерывное развертывание без DevOps?	289
10.4.1. Завершение конвейера непрерывного развертывания	290
10.5. В заключение	292
Ссылки	292
Глава 11. Непрерывное развертывание, DevOps и архитектура ПО	293
11.1. Введение	293
11.2. Архитектура программного обеспечения	293
11.2.1. Зачем нужна архитектура программного обеспечения?	294
11.3. Оптимизация архитектуры для непрерывного развертывания	296
11.3.1. Деление на мелкие единицы развертывания	297
11.4. Интерфейсы	298
11.4.1. Закон Постела, или принцип надежности	300
11.4.2. Страховка от сбоев	300
11.4.3. Состояние	301
11.5. Базы данных	302
11.5.1. Поддержание стабильности схемы базы данных	303
11.5.2. База данных = компонент	304
11.5.3. Представления и хранимые процедуры	304
11.5.4. Отдельная база данных для каждого компонента	305
11.5.5. Базы данных NoSQL	305

11.6. Микрослужбы	305
11.6.1. Микрослужбы и непрерывное развертывание.....	306
11.6.2. Внедрение непрерывного развертывания с микрослужбами	307
11.6.3. Микрослужбы способствуют внедрению непрерывного развертывания	307
11.6.4. Организация	308
11.7. Внедрение новых возможностей.....	308
11.7.1. Отдельные ветви для новых возможностей	309
11.7.2. Переключение возможностей	309
11.7.3. Преимущества	310
11.7.4. Примеры использования переключения возможностей.....	311
11.7.5. Недостатки	312
11.8. В заключение	312
Ссылки	313
Глава 12. Заключение: основные преимущества	315
Ссылки	316

Моей семье и друзьям в признательность за их поддержку.

И компьютерному сообществу за все приятные моменты.

Предисловие

П.1. Обзор методологии непрерывного развертывания и книги

Методология непрерывного развертывания (Continuous Delivery) позволяет еще быстрее и надежнее, чем прежде, передавать программное обеспечение в эксплуатацию. Основой для таких улучшений является конвейер непрерывного развертывания, в значительной степени автоматизирующий внедрение программного обеспечения и тем самым уменьшающий риски, возникающие при переходе на новые версии.

Как возник термин «непрерывное развертывание» (Continuous Delivery)?

Манифест гибкой разработки программного обеспечения (<http://agilemanifesto.org/iso/ru/manifesto.html>) определяет следующий принцип как наиболее важный:

«Наивысшим приоритетом для нас является удовлетворение потребностей заказчика благодаря регулярной и ранней поставке ценного программного обеспечения».

То есть непрерывное развертывание принадлежит к комплексу методологий гибкой разработки.

В этой книге рассказывается, как и с помощью каких технологий сконструировать такой конвейер на практике. Внимание будет уделено не только компиляции и установке, но также разнообразным тестам, гарантирующим высокое качество программного обеспечения.

Вы узнаете, как методология непрерывного развертывания воздействует на интеграцию разработки и эксплуатации (DevOps). Познакомьтесь

с влиянием непрерывного развертывания на архитектуру программного обеспечения. При этом вашему вниманию будет предложена не только теория, лежащая в основе непрерывного развертывания, но и возможные комплексы технологий, охватывающие сборку, непрерывную интеграцию, нагрузочные и приемочные тесты, а также мониторинг. Внутренние компоненты комплекса технологий всегда сопровождаются примерами проектов, которые помогут получить практические навыки. Данная книга не только предложит вам познакомиться с комплексом технологий, но также укажет направления для получения более полноценных знаний по различным темам. Выдвигая предложения для экспериментов и описывая порядок их проведения, она способствует закреплению практических знаний. Благодаря такому подходу вы получите указания, как продолжить изучение представленных тем и как развить практические навыки владения ими. Например, учебные проекты могут служить основой для самостоятельных экспериментов или даже для создания собственного конвейера непрерывного развертывания.

Дополнительную информацию, список опечаток и ссылки на примеры проектов можно найти на веб-сайте книги: <http://continuous-delivery-book.com>.

П.2. Зачем нужно непрерывное развертывание?

Зачем вообще использовать методологию непрерывного развертывания? Ответом на этот вопрос послужит короткая история, а насколько она правдива — это уже другой вопрос.

П.2.1. Короткая история

В отделе маркетинга одного предприятия — назовем его «Big Money Online Commerce Inc.» — решили пересмотреть процедуру регистрации на сайте своего интернет-магазина. Целью было привлечение новых клиентов и увеличение объема продаж. Группа программистов приступила к работе и спустя некоторое время выдала результат.

Внесенные изменения необходимо было протестировать. С этой целью коллектив «Big Money Online Commerce Inc.» создал тестовое окружение, приложив немало усилий. Тестировать программное обеспечение в этом окружении приходилось вручную. К сожалению, в ходе тестирования были

выявлены ошибки. Но к тому времени разработчики уже были заняты работой над другим проектом, и, чтобы вспомнить детали старого проекта и исправить ошибки, им потребовалось время. Кроме того, поскольку тестирование выполнялось вручную, причиной возникновения некоторых «ошибок» стало неправильное тестирование, а другие просто не воспроизводились по некоторым причинам.

Наконец исправленный код необходимо было перенести в действующее окружение. Процесс переноса был очень трудоемким, потому что веб-сайт компании «Big Money Online Commerce Inc.» развивался не первый год и имел очень сложную организацию. Выгоды от внедрения какой-то одной функции не оправдывали усилий, затрачиваемых на развертывание. По этой причине развертывание новых версий производилось один раз в месяц и планировалось заранее. В конце концов, установку изменений в процедуру регистрации можно было бы совместить с переносом изменений в других программных компонентах, произведенных за последний месяц. Соответственно, одна из ночей резервировалась для развертывания всех изменений. К сожалению, во время развертывания возникла ошибка. Коллектив приступил к анализу проблемы. Но она оказалась слишком сложной, из-за чего на следующее утро система оказалась недоступной. К тому времени разработчики устали и находились под большим психологическим давлением — каждая минута простоя системы обходилась в кругленькую сумму. Однако возврат к прежней версии был невозможен, потому что некоторые изменения, уже развернутые к тому моменту, нельзя было просто откатить. Только к концу дня, после всеобъемлющего анализа, оперативной группе удалось устранить проблему, и веб-сайт включился в работу. Как оказалось, проблема крылась в конфигурационных файлах, в которых хранились настройки для тестового окружения и которые просто забыли поменять перед переносом в рабочее окружение.

На какой-то момент все выглядело замечательно, но... обнаружилась еще одна ошибка, которую просто проглядели. Эта ошибка должна была «всплыть» в ходе ручного тестирования. Более того, тест, способный выявить эту ошибку, успешно выполнялся. Однако на этапе тестирования было найдено и исправлено множество других ошибок, а данный конкретный тест выполнялся до исправлений. Упомянутая ошибка возникла как раз из-за одного внесенного исправления, но, поскольку ручной тест не был повторен, ошибка перекечевала в рабочее окружение.

В результате на следующий день она проявилась, и регистрация на веб-сайте компании «Big Money Online Commerce Inc.» вообще перестала

работать. Этого никто не заметил, пока первый потенциальный клиент не позвонил на номер горячей линии и не сообщил о проблеме. К сожалению, в этот момент нельзя было сказать, сколько потенциальных клиентов было потеряно из-за отказа, — статистика о работе веб-сайта оказалась утрачена. Сможет ли оптимизированный процесс регистрации компенсировать потерянные регистрации — большой вопрос. Вполне возможно, что внесенные изменения вообще приведут не к увеличению регистраций, как планировалось изначально, а к уменьшению. Кроме всего прочего, новая версия оказалась значительно медленнее предыдущей — чего вообще никто не ожидал.

Вследствие произошедшего в «Big Money Online Commerce Inc.» начали очередной цикл оптимизации и усовершенствования, запланировав очередное обновление веб-сайта через месяц. Какова вероятность, что следующее развертывание пройдет удачнее?

П.2.2. Непрерывное развертывание способно помочь в таких ситуациях

Непрерывное развертывание предотвращает подобные проблемы различными мерами.

- Развертывание осуществляется чаще — вплоть до нескольких раз в день. Это сокращает время ввода в действие новых возможностей.
- Частые развертывания также ускоряют получение отзывов на новые особенности и изменения в коде. Разработчикам не приходится вспоминать, что делалось в прошлом месяце.
- Чтобы развертывание протекало быстрее, создание тестового окружения и собственно тестирование должны осуществляться автоматически, иначе на это будет уходить слишком много сил.
- Автоматизация улучшает воспроизводимость: если тестовое окружение было благополучно создано, ту же автоматизированную процедуру можно использовать для создания рабочего окружения практически с той же конфигурацией. Как следствие, проблемы, вызванные ошибками в конфигурации, не будут возникать в рабочем окружении.
- Кроме того, автоматизация дает больше гибкости. Тестовые окружения можно создавать по мере необходимости. Например, в случае изменения пользовательского интерфейса можно создать отдельное тестовое

окружение для рекламы на ограниченный период, а для всестороннего нагрузочного тестирования — отдельные окружения, близкие по своим параметрам к рабочему, и уничтожить их после тестирования, чтобы не вкладывать деньги в ненужные аппаратные средства (например, когда используются облачные технологии).

- Автоматизация тестирования упрощает воспроизведение ошибок. Поскольку во время каждого теста выполняется одна и та же последовательность действий, с процедурой выполнения теста не связано никаких ошибок.
- Автоматизированные тесты могут выполняться чаще и не требуют дополнительных усилий. Соответственно, любые исправления постоянно тестируются, и любые ошибки в процедуре регистрации на веб-сайте компании «Big Money Online Commerce Inc.» могут обнаруживаться до развертывания нового кода в рабочем окружении.
- Риски, связанные с установкой новых версий, существенно уменьшаются за счет такой настройки процедуры развертывания в рабочем окружении, которая дает возможность легко откатиться к старой версии, если это потребуется. Это позволяет предотвратить простои действующего сервера, описанные в предыдущей истории.
- Наконец, приложения, как предполагается, находятся под постоянным «присмотром» — мониторингом, поэтому неожиданная остановка любого процесса, например отвечающего за регистрацию, не останется незамеченной.

Подводя итоги, можно сказать, что непрерывное развертывание позволит бизнесу быстрее внедрять новые возможности и увеличит надежность компьютерных систем. Повышенную надежность можно рассматривать как дополнительный ресурс для разработчиков, не думаю, что имеются любители заниматься развертыванием новых версий и исправлением ошибок ночью или в выходные дни. Кроме того, гораздо лучше, если разработчики и сотрудники предприятия обнаруживают ошибки во время тестирования, а не после развертывания нового кода в рабочем окружении.

Существует большое количество технологий и приемов реализации непрерывного развертывания. Непрерывное развертывание оказывает влияние на самые разные аспекты, даже на архитектуру программного обеспечения. Об этом рассказывается в данной книге. Цель этих технологий — создать быстрый и надежный процесс переноса программного обеспечения в рабочее окружение.

П.3. Кому адресована эта книга?

Эта книга адресована руководителям, архитекторам, разработчикам и администраторам, желающим внедрить методологию непрерывного развертывания как метод и (или) средство интеграции разработки и эксплуатации (DevOps) в своей организации.

- В теоретической части книги руководители познакомятся с процессом, лежащим в основе непрерывного развертывания, а также с требованиями и выгодами для их предприятий. Кроме того, они научатся оценивать технические последствия от внедрения непрерывного развертывания.
- Разработчикам и администраторам будет представлено исчерпывающее введение в технические аспекты, благодаря чему они смогут приобрести навыки, необходимые для конструирования и внедрения конвейера непрерывного развертывания.
- Архитекторы, помимо технических аспектов, узнают также о влиянии непрерывного развертывания на архитектуру программного обеспечения (глава 11).

Книга знакомит с разными технологиями реализации непрерывного развертывания. Примером служит проект на языке Java. Для некоторых областей — например, реализации приемочных испытаний — будут представлены технологии для других языков программирования. В книге упоминаются подобные альтернативы, но основное внимание уделяется языку Java. Технологии автоматизированного распределения инфраструктуры не зависят от используемого языка программирования. Книга особенно пригодится читателям, активно использующим Java; адаптацией представленных технологий для других языков читателям придется заниматься самим.

П.4. Краткое содержание глав

Книга делится на три части. Первая часть знакомит с основами, необходимыми для понимания методологии непрерывного развертывания.

- Глава 1, «Непрерывное развертывание: что и как?», знакомит с терминологией непрерывного развертывания, а также объясняет, как и какие проблемы она решает. Кратко знакомит с конвейером непрерывного развертывания.

- Непрерывное развертывание требует автоматизации развертывания инфраструктуры — установки программного обеспечения на серверы. Глава 2, «Подготовка инфраструктуры», знакомит с некоторыми подходами к решению этой задачи. Для автоматизации установки программного обеспечения можно использовать Chef. Для развертывания тестового окружения на компьютерах разработчиков можно использовать Vagrant. Docker — не только очень эффективное решение виртуализации, он также может служить средством автоматизации установки программного обеспечения. В конце первой части дается краткий обзор использования облачных решений PaaS (Platform as a Service — платформа как услуга) для организации непрерывного развертывания.

Вторая часть включает главы, детально описывающие разные компоненты конвейера непрерывного развертывания. После знакомства с основными идеями будут представлены примеры конкретных технологий, которые используются для реализации обсуждаемых компонентов конвейера.

- Глава 3, «Автоматизация сборки и непрерывная интеграция», написанная Бастианом Спаннебергом (Bastian Spanneberg), описывает происходящее в процессе передачи новой версии программного обеспечения. Дается краткое введение в инструменты сборки, такие как Gradle или Maven, и модульное тестирование, а также обсуждается технология непрерывной интеграции (Continuous Integration) с применением Jenkins. Далее следует обзор SonarQube, инструмента статического анализа и проверки качества программного кода, а также репозиториях, таких как Nexus или Artifactory.
- Глава 4, «Приемочные тесты», знакомит с использованием JBehave и Selenium для автоматизации приемочных испытаний.
- Глава 5, «Тестирование пропускной способности», посвящена исключительно теме пропускной способности. Роль примера в ней играет технология Gatling.
- Глава 6, «Исследовательское тестирование», рассматривает особенности испытаний вручную с целью проверки новых возможностей и выявления общих проблем в приложениях.

Главы, перечисленные выше, описывают начальные этапы конвейера непрерывного развертывания. Они оказывают основное влияние на разработку программного обеспечения. Следующие главы знакомят с технологиями и приемами в тех областях непрерывного развертывания, которые ближе к рабочему окружению.

- Глава 7, «Развертывание — ввод в эксплуатацию», описывает подходы, помогающие минимизировать риски, которые могут возникнуть в процессе развертывания приложения в рабочем окружении.
- В процессе работы приложение собирает разные данные, характеризующие особенности его функционирования. Глава 8, «Эксплуатация», знакомит с технологиями, упрощающими сбор и анализ таких данных: стек ELK (Elasticsearch-Logstash-Kibana) — для анализа файлов журналов и Graphite — для мониторинга.

Для технологий, описываемых в этих главах, приводятся примеры, которые читатель сможет опробовать на собственном компьютере и поэкспериментировать с ними, чтобы получить практический опыт. Благодаря автоматизированной инфраструктуре запуск этих примеров не составит труда.

Наконец, возникает вопрос: как внедрить методологию непрерывного развертывания и какой эффект это даст? Эти вопросы обсуждаются в третьей части книги.

- Глава 9, «Внедрение методологии непрерывного развертывания на предприятии», показывает, как можно внедрить методологию непрерывного развертывания в своей организации.
- Глава 10, «Непрерывное развертывание и DevOps», описывает объединение разработки (Development, Dev) и эксплуатации (Operations, Ops) в один организационный модуль (DevOps).
- Непрерывное развертывание также оказывает влияние на архитектуру приложений. Связанные с этим вопросы обсуждаются в главе 11, «Непрерывное развертывание, DevOps и архитектура ПО».
- Завершается книга главой 12, «Заключение: основные преимущества».

П.5. Как читать эту книгу

Главы книги можно читать в разном порядке. В этом разделе описывается несколько возможных вариантов (рис. П.1). Введение в главе 1 будет интересно всем читателям — глава кратко знакомит с основными терминами и рассказывает о побудительных мотивах внедрения непрерывного развертывания.

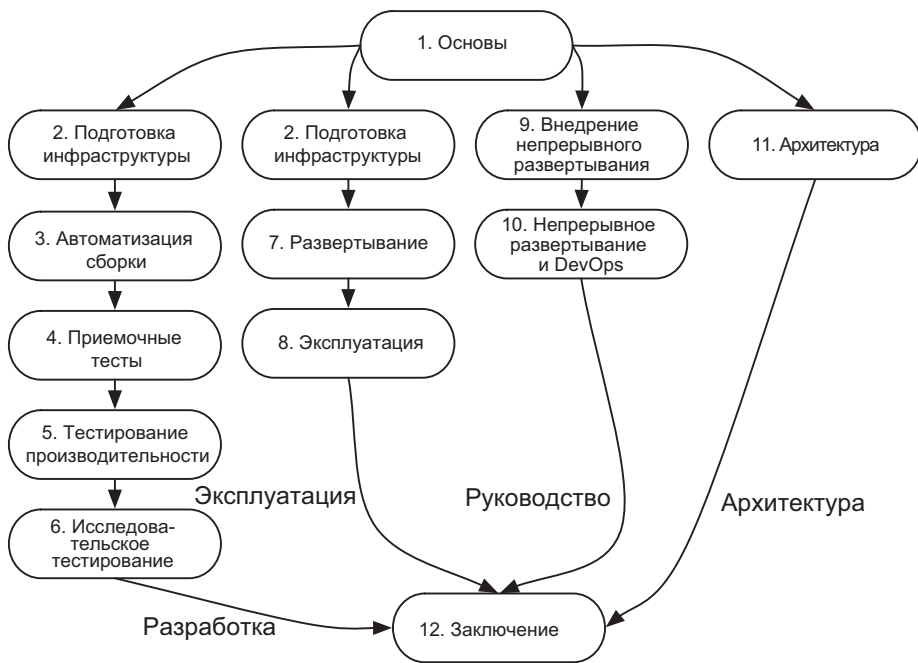


Рис. П.1. Порядок чтения глав книги

Главы предназначены для разных целевых аудиторий.

- Для технических специалистов, главным образом занимающихся разработкой, наибольший интерес представляют главы, описывающие разработку и тестирование. Помимо аспектов разработки в этих главах обсуждаются проблемы контроля качества и сборки и демонстрируется, как эти задачи решаются в методологии непрерывного развертывания. Кроме того, в них приводится конкретный код и примеры применения технологий, взятые из экосистемы Java.
- Администраторам и специалистам, занимающимся эксплуатацией, особенно интересны будут такие темы, как развертывание, распределение инфраструктуры и эксплуатация, на которые непрерывное развертывание оказывает существенное влияние.
- Для руководителей наибольший интерес представляет вводная информация и взаимосвязь между методологиями интеграции разработки и эксплуатации и непрерывным развертыванием. Эти две главы показывают, как непрерывное развертывание влияет на организации.

- Архитекторам свойствен более широкий взгляд на вещи. Основной интерес для них представляет глава 11, где обсуждается тема влияния непрерывного развертывания на архитектуру программного обеспечения; однако, поскольку им важно иметь представление о технических деталях и оценивать ситуацию с точки зрения управления, они могут также прочитать, хотя бы выборочно, главы, посвященные этим темам.
- Наконец, последняя глава посвящена заключительным выводам.

ПРИМЕЧАНИЕ

Зарегистрируйте свой экземпляр книги «Практическое руководство по непрерывному развертыванию» на сайте informit.com, чтобы получить доступ к загружаемым материалам, обновлениям и корректировкам. Для этого перейдите по адресу informit.com/register и войдите со своей учетной записью или создайте новую. Введите код книги ISBN 9780134691473 и щелкните на кнопке Submit (Отправить). После регистрации вы найдете дополнительную информацию в разделе «Registered Products» («Зарегистрированные продукты»).

Благодарности

Я хотел бы сказать спасибо Бастиану Спаннебергу (Bastian Spanneberg) за его вклад в эту книгу. Также я выражаю благодарность рецензентам — их неоценимые комментарии оказали существенное влияние на эту книгу: Марсель Биркнер (Marcel Birkner), Ларс Генч (Lars Gentsch), Халил-Джем Гюрсой (Halil-Gem Gürsoy), Феликс Мюллер (Felix Müller), Саша Мёллеринг (Sascha Möllering) и Александр Папаспироу (Alexander Papaspyrou). Кроме того, хочу сказать спасибо моим коллегам за продолжительные обсуждения — в ходе них родилось немало идей.

Спасибо моим работодателям из innoQ.

Наконец, я хочу выразить свою благодарность моим друзьям, родителям и родственникам, общением с которыми я часто пренебрегал, пока работал над книгой, и особенно моей супруге, выполнившей перевод книги на английский язык.

Конечно же, спасибо всем тем, кто помогал создавать технологии, описываемые в книге, и своим трудом сделал возможным непрерывное развертывание.

И последнее, но не менее важное: я хочу поблагодарить dpunkt.verlag и Рене Шёнфельдта (René Schönfeldt), оказывавшим профессиональную поддержку во время работы над немецкой версией книги.

Издательство Addison-Wesley любезно предоставило мне возможность опубликовать версию книги на английском языке. Огромную поддержку в этом мне оказали: Крис Зан (Chris Zahn), Крис Гузиковски (Chris Guzickowski), Лори Лайонс (Lori Lyons), Даянидхи Карунанидхи (Dhayanidhi Karunanidhi) и Ларри Салки (Larry Sulky).

Об авторе

Эберхард Вольф (Eberhard Wolff), сотрудник компании innoQ в Германии, имеет более чем 15-летний опыт разработки архитектур программного обеспечения и консультирования по вопросам, лежащим на стыке предпринимательства и технологий. Он неоднократно выступал с докладами на международных конференциях, был членом программных комитетов на нескольких конференциях и написал более 100 статей и книг. Круг его профессиональных интересов включает разработку современных архитектур — часто с привлечением облачных технологий, непрерывного развертывания, интеграции разработки и эксплуатации (DevOps), микрослужб и NoSQL.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу *comp@piter.com* (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства *www.piter.com* вы найдете подробную информацию о наших книгах.

Часть I

ОСНОВЫ

В этой части глава 1 знакомит с основами методологии непрерывного развертывания. Глава 2 разъясняет технические аспекты непрерывного развертывания: в ней обсуждается автоматизация развертывания инфраструктуры и установки программного обеспечения, без которых реализация непрерывного развертывания невозможна.

1

Непрерывное развертывание: что и как?

1.1. Введение: что такое непрерывное развертывание?

Ответить на этот вопрос совсем непросто. Основоположники этого термина не дали четкого определения [1]. Мартин Фаулер (Martin Fowler) в своем обсуждении [2] непрерывного развертывания (Continuous Delivery) особо подчеркивает возможность в любой момент развернуть программное обеспечение (ПО) в рабочем окружении. Это требует автоматизации процессов, необходимых для установки ПО и оценки его качества. С другой стороны, в «Википедии» [3] «непрерывное развертывание» определяется как процесс оптимизации и автоматизации выпуска новых версий ПО.

Наконец, главная цель методологии непрерывного развертывания заключается в анализе и оптимизации процесса, конечным пунктом которого является выпуск программного обеспечения. Строго говоря, этот процесс часто размывается во время разработки.

1.2. Почему процесс выпуска ПО настолько сложен

Выпуск новых версий ПО сопряжен с немалыми сложностями — практически каждому разработчику приходилось проводить выходные в офисе, занимаясь развертыванием новой версии ПО в рабочем окружении. Такие события в конечном итоге заканчиваются установкой новой версии, потому что путь назад, к предыдущей версии, с некоторого момента оказывается еще более тернистым и опасным, чем путь вперед. Однако за установкой часто следует продолжительный этап стабилизации обновленной версии.

1.2.1. Непрерывная интеграция вселяет надежду

В настоящее время наибольшие сложности доставляет только процедура развертывания ПО в рабочем окружении. Но совсем недавно проблемы начинались намного раньше: отдельные группы работали над своими модулями независимо друг от друга и перед выпуском разных версий должны были сначала интегрировать их вместе. В процессе интеграции, когда модули впервые объединялись друг с другом, система часто даже не компилировалась. Порой приходилось тратить дни и даже недели, пока не удавалось интегрировать все изменения и добиться успешной компиляции. Только после этого можно было начать этап развертывания. К настоящему времени эти проблемы практически решены: все группы работают с общей версией кода, который непрерывно и автоматически интегрируется, компилируется и тестируется. Этот подход называется непрерывной интеграцией (Continuous Integration). Инфраструктура, необходимая для непрерывной интеграции, детально описывается в главе 3, «Автоматизация сборки и непрерывная интеграция». Благополучное решение проблем, связанных с этим этапом, вселяет надежду, что так же будут решены проблемы, возникающие на других этапах пути к развертыванию ПО в рабочем окружении.

1.2.2. Медленные и опасные процессы

Процессы, происходящие на более поздних этапах, зачастую сложны и трудоемки, а ручные операции делают их чрезвычайно утомительными и способствуют появлению ошибок. Это верно не только для этапа развертывания, но также для предыдущих этапов — например, тестирования. А если ручные операции выполняются лишь несколько раз в год, это только увеличивает вероятность ошибок и риск провала всей процедуры.

Из-за высоких рисков и большой сложности новые версии внедряются в эксплуатацию все реже. А отсутствие практики замедляет этот процесс еще больше. Кроме того, это мешает оптимизации процесса.

1.2.3. Есть возможность ускорения

С другой стороны, всегда есть возможность быстро внедрить новую версию, особенно в чрезвычайной ситуации, например, когда нужно срочно

исправить ошибку. Однако в этом случае пропускаются этапы тестирования и, соответственно, отбрасываются меры безопасности, являющиеся неотъемлемой частью стандартного процесса. Это очень высокий риск, потому что необходимость тестирования обусловлена весьма вескими причинами.

Итак, стандартный путь внедрения новой версии тернист и труден. В чрезвычайных ситуациях этот путь можно пройти быстрее, но ценой еще более высокого риска.

1.3. Ценность непрерывного развертывания

Хотелось бы на основе преимуществ и подходов непрерывного развертывания оптимизировать процедуру внедрения новых версий.

Фундаментальный принцип непрерывной интеграции гласит: «Если что-то доставляет боль, делай это чаще». Звучит издевательски, но на самом деле это подход к решению проблем. В качестве средства ухода от проблем, связанных с выпуском новых версий, старайтесь внедрять новые версии как можно быстрее. Эта процедура должна выполняться как можно чаще и как можно раньше, чтобы быстрее можно было достичь высокой скорости и надежности ее выполнения. Как следствие, методология непрерывного развертывания вынуждает организацию принять и освоить новый стиль работы.

В этом подходе нет ничего необычного: как отмечалось выше, любое IT-подразделение способно быстро переносить исправления в рабочее окружение, и в таких ситуациях часто принято выполнять только часть тестов и проверок на безопасность. Это допустимо, потому что изменения малы и не влекут больших рисков. Существует другой подход, минимизирующий риски: не пытаться защититься от проблем, применяя сложные процедуры и реже выпуская новые версии, а наоборот, чаще выпускать небольшие изменения. Этот подход по своей сути идентичен стратегии непрерывной интеграции: непрерывная интеграция подразумевает регулярное объединение даже самых небольших изменений, внесенных отдельными разработчиками или группами, вместо накопления изменений в течение дней и недель с последующим их объединением в конце — стратегии, часто вызывающей существенные проблемы; в некоторых случаях

проблемы оказываются настолько велики, что программное обеспечение даже не компилируется.

Однако непрерывное развертывание — это больше, чем «часто и небольшими порциями». Непрерывное развертывание основывается на разных ценностях, исходя из которых могут предприниматься конкретные меры технического характера.

1.3.1. Регулярность

Под регулярностью понимается более частое выполнение процедур. Все процедуры, необходимые для внедрения ПО в эксплуатацию, должны выполняться регулярно, а не только когда выпускается новая версия. Например, обязательно должны создаваться испытательное окружение и окружение для обкатки. Тестовое окружение можно использовать для проведения технических и приемочных тестов. Окружение для обкатки может использоваться конечным заказчиком для проверки и оценки возможностей новой версии. Для организации этих окружений процедуру их создания можно превратить в обычный процесс, который используется не только для создания рабочего окружения. Чтобы создание множества окружений не требовало значительных усилий, процедуры должны быть максимально автоматизированными. Регулярность обычно приводит к необходимости автоматизации. Все то же самое относится к тестам: нет смысла откладывать тестирование до момента выпуска новой версии — тесты должны выполняться регулярно. В этом случае регулярность также вынуждает максимально автоматизировать процедуру тестирования, чтобы уменьшить прилагаемые усилия. Кроме того, регулярность способствует высокой надежности — процедуры, выполняющиеся часто, могут повторяться и выполняться с высокой степенью надежности.

1.3.2. Контролируемость

Все изменения в программном обеспечении, которые предполагается перенести в рабочее окружение и в инфраструктуру, необходимую для новой версии, должны быть контролируемыми. Всегда должна существовать возможность восстановить ПО и инфраструктуру в том или ином состоянии. Это влечет за собой необходимость поддержки версионирования, причем

не только программного обеспечения, но также необходимых окружений. В идеале должна существовать возможность восстановить каждое состояние ПО вместе с окружением и правильными настройками, необходимыми для его работы. То есть все изменения в программном обеспечении и окружениях должны контролироваться. Это позволит легко создать подходящую систему для анализа ошибок. И наконец, такой подход дает возможность документировать изменения и исследовать их.

Одно из возможных решений данной задачи — ограничение доступа к рабочему окружению и окружению для обкатки определенному кругу лиц. Это, как предполагается, поможет избежать «исправлений на скорую руку», которые не документируются и не контролируются. Кроме того, требования к безопасности и защищенности данных вообще запрещают доступ к рабочим окружениям.

При использовании методологии непрерывного развертывания внедрение в окружение возможно только при изменении сценария установки. Изменения в сценариях становятся контролируемыми, когда они попадают в систему управления версиями. Разработчики сценариев также не должны иметь доступа к данным в рабочем окружении и, соответственно, не должны быть источником проблем с защищенностью этих данных.

1.3.3. Регрессии

Для минимизации рисков, связанных с внедрением в эксплуатацию новой версии программного обеспечения, это программное обеспечение необходимо протестировать. Несомненно, правильная работа новых функций должна проверяться на этапе тестирования, но нередко приходится прилагать значительные усилия, чтобы избежать регрессий — ошибок, возникающих в результате изменения уже протестированных фрагментов программного кода. Для этого следует выполнять все имеющиеся тесты, потому что изменения в одном месте системы могут вызвать ошибки в другом. Этот подход требует автоматизации тестирования. Иначе тестирование может отнять слишком много сил и времени. Если какая-то ошибка все же просочится в рабочее окружение, ее можно будет обнаружить в процессе мониторинга. В идеале должна существовать возможность максимально просто вернуть (откатить) рабочее окружение к предыдущей версии, не содержащей ошибок, или внести небольшие исправления (накатить). Идея заключается в том, чтобы иметь некоторый аналог системы раннего обнаружения, помогающей вскрывать и решать проблемы на разных этапах разработки проекта, таких как тестирование и развертывание.

1.4. Преимущества непрерывного развертывания

Непрерывное развертывание предлагает множество преимуществ, ценность которых, однако, зависит от используемого сценария, — как следствие все это будет влиять на характер реализации непрерывного развертывания.

1.4.1. Непрерывное развертывание сокращает сроки внедрения

Непрерывное развертывание сокращает сроки, необходимые для внедрения изменений в эксплуатацию. Это дает существенные преимущества для бизнеса: он получает возможность быстрее реагировать на изменения рынка.

Однако скорость реакции на изменения рынка — не единственное преимущество: современные подходы, такие как Lean Startup¹ [4], пропагандируют стратегии, помогающие получать дополнительные выгоды от сокращения сроков внедрения. Методология Lean Startup заключается в размещении продукта на рынке и оценке его востребованности с минимальными затратами. По аналогии с научным экспериментом заранее выдвигается гипотеза об успехе продукта на рынке. Затем выполняется эксперимент, и по его результатам оценивается успех или неудача.

1.4.2. Один пример

Давайте рассмотрим конкретный пример. В интернет-магазине появляется новая услуга: возможность выбора определенной даты доставки товара. В качестве эксперимента можно дать объявление, рекламирующее новую услугу. Оценкой успеха в этом эксперименте могло бы служить количество щелчков на ссылке внутри объявления. В этот момент необходимое программное обеспечение еще не реализовано, то есть рекламируемая услуга пока недоступна. Если эксперимент не даст обнадеживающего результата, услуга будет оценена как бесперспективная и приоритет может быть отдан другим услугам — и все это практически без затрат.

¹ Переводится как «бережливый стартап» (https://ru.wikipedia.org/wiki/Бережливый_стартап). — Примеч. пер.

1.4.3. Реализация услуги и ее внедрение в эксплуатацию

Если эксперимент увенчался успехом, услуга будет реализована и внедрена в эксплуатацию. Даже этот шаг можно реализовать как эксперимент: измерить показатели, помогающие оценить успех новой услуги, например количество заказов с фиксированной датой доставки.

1.4.4. Следующий шаг

В ходе анализа результатов измерений выяснилось, что количество заказов достаточно высоко, но самое интересное — большинство таких заказов доставляется не заказчикам непосредственно, а третьим лицам. Дополнительные исследования помогли выяснить, что заказанные товары в этих случаях являются подарками ко дню рождения. Исходя из этой информации услугу можно расширить — например, добавить календарь для выбора дня рождения и рекомендации по выбору подарков. Конечно, это требует проектирования дополнительных возможностей, их реализации, внедрения и, наконец, оценки успеха. В качестве альтернативы можно попробовать оценить успех будущих возможностей до их реализации — через рекламу, опросы, социологические исследования и другими способами.

1.4.5. Непрерывное развертывание дает конкурентные преимущества

Непрерывное развертывание позволяет быстрее вводить в эксплуатацию необходимые изменения в программном обеспечении, а предприятиям — быстро проверять разные идеи и определять перспективные направления развития бизнеса. Это создает конкурентные преимущества: чем больше идей можно проверить, тем проще отфильтровать ошибочные и оставить верные — и это не на основе субъективных оценок, а на объективных показателях (рис. 1.1).

1.4.6. Без непрерывного развертывания

Без непрерывного развертывания внедрение услуги доставки заказа в определенную дату пришлось переносить на момент внедрения следующей версии системы, до выхода которой может пройти несколько месяцев. До ее

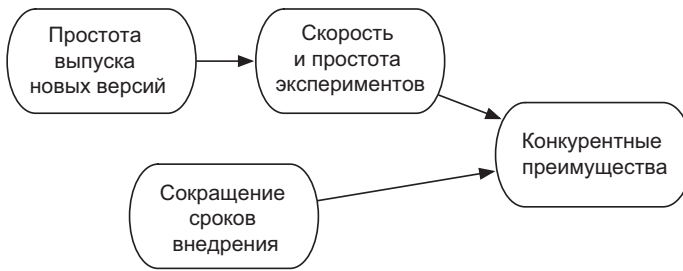


Рис. 1.1. Причины использовать методологию непрерывного развертывания

выпуска отдел рекламы едва ли осмелился бы объявить об услуге, потому что долгий срок внедрения делает это бессмысленным. Если услуга не будет пользоваться спросом, затраты на ее внедрение окажутся напрасными. Оценить успех новой услуги, без сомнений, можно классическим способом; однако среагировать на оценку удастся лишь с большим опозданием. Дополнительные услуги, например связанные с покупкой подарков на день рождения, появятся на рынке намного позже, поскольку для их внедрения придется ждать выхода следующей версии системы и запуска очередного продолжительного цикла разработки. Кроме того, остаются сомнения в надежности анализа успеха новой услуги, чтобы на его основе определить потенциал дополнительных услуг, поддерживающих покупку подарков на день рождения.

1.4.7. Непрерывное развертывание и Lean Startup

Итак, благодаря непрерывному развертыванию циклы оптимизации выполняются намного быстрее, потому что любую новую функцию можно внедрить практически в любой момент. Это позволяет применять такие подходы, как Lean Startup, и тем самым влиять на развитие бизнеса: быстрее вводить новые услуги — не ограничиваться длительными периодами планирования, а практически мгновенно реагировать на результаты текущих экспериментов. Это достаточно просто реализовать на начальных этапах развития предприятия, но подобные структуры можно также выстроить в классических организациях. Подход Lean Startup, к сожалению, получил немного неточное название: он используется для вывода на рынок новых продуктов через последовательность экспериментов, и, конечно же, его можно применять не только в стартапах, но и на классических предприятиях. Его также можно применять, когда продукт распространяется

другими способами, например на носителях, таких как компакт-диски, или как часть другого продукта, например автомобиля. В таких ситуациях установка программного продукта должна быть максимально упрощена и автоматизирована. Помимо круга клиентов в подобных случаях следует определить тех, кто пожелает заняться тестированием новых версий программного обеспечения и присылать свои отзывы, то есть обычных бета-тестеров или просто опытных пользователей.

1.4.8. Влияние непрерывного развертывания

Непрерывное развертывание влияет на процесс разработки программного обеспечения: он должен поддерживать внедрение в эксплуатацию новых возможностей по мере их появления. Некоторые процессы разработки основаны на итерациях длительностью от одной до нескольких недель. В конце каждой итерации в эксплуатацию вводится новая версия с несколькими новыми возможностями. Это не самый лучший вариант для непрерывного развертывания, потому что при нем новые возможности не могут перемещаться по этапам разработки независимо друг от друга. Итеративная организация процесса также препятствует применению подхода Lean Startup: когда одновременно выпускается сразу несколько новых возможностей/услуг, трудно оценить, какая из них больше повлияла на измеряемые оценки успеха. Представьте, что услуга доставки товара в определенную дату была введена одновременно со снижением стоимости доставки — мы не сможем определить, какое из двух нововведений оказало большее влияние на увеличение объемов продаж.

То есть такие процессы разработки, как Scrum, XP (Extreme Programming — экстремальное программирование) и, конечно, обычная каскадная (waterfall) модель, препятствуют использованию непрерывного развертывания, потому что всегда предполагают выпуск новых версий сразу с несколькими новыми возможностями. Методология Kanban [5], напротив, нацелена на продвижение единственной возможности через различные этапы разработки. Она идеально соответствует методологии непрерывного развертывания. Конечно, другие процессы также можно изменять по отдельности для поддержки развертывания новых возможностей, но после такой адаптации их реализация уже не будет соответствовать описанию в многочисленных книгах. Существует еще один вариант — изначально деактивировать дополнительные функции при внедрении в эксплуатацию нескольких новых возможностей, чтобы измерить и оценить их эффект по отдельности.

Наконец, этот подход также предполагает исполнение коллективами нескольких разных ролей. Помимо разработки и эксплуатации программного обеспечения они должны также взять на себя отдельные бизнес-роли, такие как маркетинг. Благодаря устранению организационных барьеров обратная связь со стороны бизнеса может воплощаться в эксперименты еще быстрее.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ Соберите информацию о методологиях Lean Startup и Kanban. Где родилась методология Kanban?

Выберите известный вам проект или одну из функций этого проекта.

- ▲ Как мог бы выглядеть минимально возможный продукт? Минимальный продукт должен давать представление о перспективах полного продукта на рынке.
- ▲ Можно ли оценить продукт без применения программного обеспечения? Например, можно ли рекламировать его другими путями? Насколько ценными являются отзывы потенциальных пользователей?
- ▲ Как оценить успех новой услуги? Можно ли, например, измерить ее влияние на объем продаж, подсчитать количество щелчков или какое-то другое значение?
- ▲ Сколько времени обычно закладывается в планы на подготовку и выпуск продукта или услуги? Как это согласуется с идеей Lean Startup?

1.4.9. Непрерывное развертывание для минимизации рисков

Как описывалось в предыдущем разделе, использование непрерывного развертывания предполагает определенную модель ведения бизнеса. Однако на классических предприятиях бизнес часто зависит от долгосрочного планирования. В таком случае подход Lean Startup не может быть реализован. Кроме того, для многих предприятий скорость внедрения не является решающим фактором. Не на всех рынках скорость является конкурентным преимуществом. Конечно, ситуация может меняться, когда та-

кие компании неожиданно сталкиваются с конкурентами, выходящими на рынок с моделью Lean Startup.

Во многих сценариях скорость внедрения не является мотивирующим условием внедрения методологии непрерывного развертывания. И все же она может оказаться полезной, потому что позволяет получать дополнительные выгоды.

- Развертывание новых версий вручную требует значительных усилий. Нередко целые отделы ИТ вынуждены проводить выходные на рабочих местах, занимаясь развертыванием новой версии. И после развертывания нередко приходится выполнять дополнительные, сопутствующие работы.
- Также высок риск неудачи: успех внедрения новой версии зависит от многих изменений, произведенных вручную, в которых легко допустить ошибку. Если ошибка не была обнаружена и устранена вовремя, она может иметь далеко идущие последствия для предприятия.

Больше всего от этого страдают сотрудники отделов ИТ: разработчики и системные администраторы, которые вынуждены по ночам и выходным заниматься вводом в эксплуатацию новых версий и исправлением ошибок. Кроме того, им приходится долгие часы работать под большим психологическим давлением из-за высоких рисков. И эти риски не следует недооценивать: компания Knight Capital, например, потеряла 440 миллионов долларов из-за неудачной попытки развернуть новую версию ПО [6]. Как следствие, она обанкротилась. В таких сценариях возникает большое количество вопросов [7], в частности почему возникла проблема, почему она не была выявлена вовремя и вообще как предотвратить такие события в других окружениях.

Решением в подобных ситуациях может стать непрерывное развертывание благодаря своим фундаментальным аспектам: высокой надежности и качеству процесса развертывания. Применение этой методологии позволяет разработчикам и системным администраторам спать спокойно в прямом смысле слова. Это обеспечивается несколькими разными факторами.

- Процессы развертывания имеют высокий уровень автоматизации, поэтому результаты легко воспроизвести. То есть после развертывания и проверки системы в тестовом окружении и в окружении для обкатки, тот же результат будет получен в рабочем окружении, потому что сами окружения полностью идентичны. Это позволяет практически

полностью устранять любые ошибки и, соответственно, уменьшать риски.

- Кроме того, сам процесс тестирования программного обеспечения становится более простым благодаря автоматизации их выполнения. Это еще больше повышает качество, потому что тестирование может выполняться намного чаще.
- Чем чаще производится развертывание новых версий, тем ниже риски, потому что в каждом случае развертывается меньшее количество изменений и нововведений. Чем меньше изменений попадает в рабочее окружение, тем ниже риск появления ошибок.

В некотором смысле ситуация выглядит парадоксальной: классическая модель разработки вынуждена развертывать новые версии в рабочем окружении как можно реже, так как с этим процессом связаны высокие риски. В каждой новой версии может появиться ошибка, влекущая потенциально пагубные последствия. Уменьшение частоты выпуска новых версий, таким образом, должно привести к уменьшению количества проблем.

Методология непрерывного развертывания, напротив, призывает увеличить частоту выпуска новых версий. В этом случае в каждой версии оказывается меньше изменений, что также способствует снижению вероятности появления ошибок. Автоматизация и надежность процессов являются главными предпосылками этой стратегии. Без этого увеличение частоты выпуска новых версий приведет к увеличению нагрузки на персонал, выполняющий процессы вручную, и рисков, возникающих из-за высокой вероятности допустить ошибку в ручном процессе. Поэтому, чтобы снизить риски, нужно стремиться не к уменьшению частоты выпуска новых версий, а к автоматизации процессов. Дополнительный бонус: увеличение частоты выпуска версий позволяет включать меньше изменений, соответственно уменьшает риск появления ошибок.

На рис. 1.2 показано, насколько мотивация непрерывного развертывания отличается от мотивации, лежащей в основе идеи Lean Startup: увеличение надежности и технического качества новых версий, а не сокращение времени ввода в эксплуатацию. А выгоды от этого получает не только бизнес, но и сотрудники отделов IT.

Поскольку преимущества столь многочисленны, возможны различные компромиссы: например, вложения в конвейер непрерывного развертывания часто окупаются, даже если тот не достигает рабочего окружения, то есть в том случае, когда рабочее окружение все еще приходится констру-

ировать вручную. В конце концов, рабочее окружение требуется создать всего один раз для каждой версии, но для нужд тестирования может понадобиться создать несколько тестовых окружений. Однако если главным движущим мотивом внедрения непрерывного развертывания все же является уменьшение сроков вывода на рынок, тогда очень важно, чтобы конвейер включал рабочее окружение.

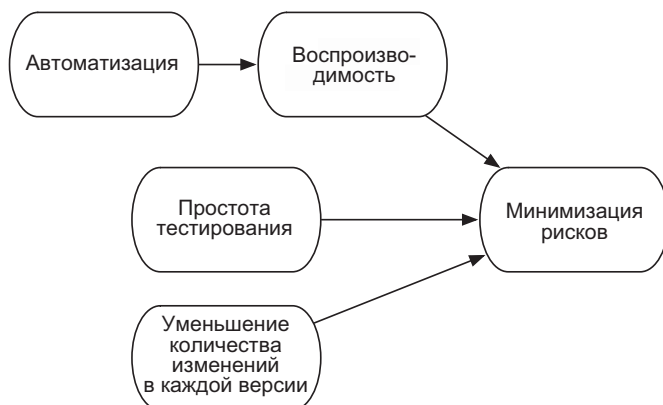


Рис. 1.2. Мотивация использования методологии непрерывного развертывания на предприятии

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Исследуйте свой текущий проект.

- ▲ Какие проблемы обычно возникают в ходе установки?
- ▲ Решит ли эти проблемы автоматизация?
- ▲ Какие текущие подходы можно упростить, чтобы облегчить автоматизацию и оптимизацию? Оцените предполагаемые усилия и ожидаемые преимущества.
- ▲ Как в настоящее время создаются рабочие и тестовые системы? Эту работу выполняет одна и та же группа? Можно ли автоматизировать оба процесса или только один?
- ▲ Для каких систем могла бы пригодиться автоматизация? Как часто создаются новые системы?

1.4.10. Быстрая обратная связь и низкие затраты

Изменяя код, разработчик получает обратную связь от собственных тестов, интеграционных тестов, тестов производительности и, наконец, из рабочего окружения. Если изменения вводятся в эксплуатацию раз в квартал, может пройти несколько месяцев между изменением кода и получением обратной связи из рабочего окружения. То же относится к приемочным тестам и тестам производительности. Если ошибка выявится потом, разработчику придется вспоминать, что он делал несколько месяцев тому назад и где может скрываться проблема.

С использованием методологии непрерывного развертывания обратная связь срабатывает намного быстрее: каждый раз, когда код движется по конвейеру, разработчик и все члены команды получают отклик. Автоматизированные приемочные тесты и тесты пропускной способности могут выполняться после каждого изменения. Это позволяет разработчику и другим членам группы быстро обнаруживать и исправлять ошибки. Скорость обратной связи можно увеличить еще больше, внедрив быстрые методики тестирования, такие как модульные тесты, и проводя тестирование сначала «в ширину» и только потом «в глубину». Это с самого начала гарантирует успешное выполнение всех функций, по крайней мере в простых случаях, и выявление ошибок на самых ранних стадиях. Кроме того, вначале должны выполняться тесты, которые, как показывает опыт, чаще терпят неудачу.

Непрерывное развертывание также способствует экономии. Любые расходы, не оплачиваемые клиентом, должны рассматриваться как напрасные траты. Любые изменения в коде являются напрасными тратами, пока они не будут развернуты в рабочем окружении, потому что только после этого клиенты начнут оплачивать изменения. Кроме того, непрерывное развертывание реализует более короткие циклы и ускоряет получение обратной связи — еще один аспект экономии.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Исследуйте свой текущий проект.

- ▲ Сколько времени проходит между изменением кода и:
 - получением обратной связи от сервера непрерывной интеграции?

- получением обратной связи с результатами приемочных тестов?
- получением обратной связи с результатами тестирования производительности/пропускной способности?
- внедрением в эксплуатацию?

1.5. Создание и организация конвейера непрерывного развертывания

Как уже отмечалось выше, методология непрерывного развертывания расширяет подход непрерывной интеграции на дополнительные этапы, как показано на рис. 1.3.

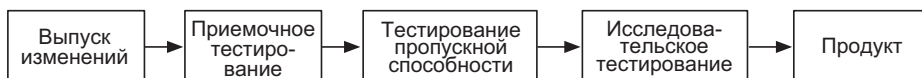


Рис. 1.3. Этапы конвейера непрерывного развертывания

Этот раздел знакомит со структурой окружения с поддержкой непрерывного развертывания. Она была предложена Джезом Хамблом¹ с соавтором и включает следующие этапы.

- Этап выпуска изменений включает действия, обычно выполняемые в рамках методологии непрерывной интеграции, такие как сборка, модульное тестирование и статический анализ кода. Подробнее этот этап конвейера обсуждается в главе 3.
- Следующий этап — приемочное тестирование, который рассматривается в главе 4, «Приемочные тесты». Строго говоря, эта тема касается автоматизированного тестирования: взаимодействия с графическим интерфейсом пользователя автоматизируются тестовой системой или необходимые операции описываются на естественном языке так, чтобы

¹ Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9. (Джез Хамбл, Дейвид Фарли. Непрерывное развертывание ПО: автоматизация процессов сборки, тестирования и внедрения новых версий программ (Signature Series). М.: Вильямс, 2011. ISBN: 978-5-8459-1739-3. — Примеч. пер.).

их можно было выполнять в автоматическом режиме. На данном этапе требуется сгенерировать необходимые тестовые окружения для выполнения приложения, если этого не было сделано раньше. Вопросам создания таких тестовых окружений посвящена глава 2, «Подготовка инфраструктуры».

- Тестирование пропускной способности (глава 5, «Тестирование пропускной способности») помогает убедиться, что программное обеспечение справится с ожидаемой нагрузкой. Для этого должны использоваться автоматизированные тесты, однозначно определяющие достаточную пропускную способность и скорость работы программного обеспечения. Под пропускной способностью подразумевается не только производительность, но и масштабируемость. Поэтому эти тесты также могут выполняться в окружении, не соответствующем рабочему. Однако тестовое окружение должно давать надежные оценки возможного поведения программного обеспечения в рабочем окружении. В зависимости от конкретных условий на этом этапе также может автоматически проверяться соответствие другим, нефункциональным требованиям, таким как безопасность и защищенность.
- В ходе исследовательского тестирования (глава 6, «Исследовательское тестирование») выполняется проверка работы приложения без какого-то строгого плана тестирования. На этом этапе эксперты в предметной области тестируют приложение, обращая внимание на новые возможности и неожиданные отклонения в поведении. То есть даже при использовании методологии непрерывного развертывания не все тесты должны быть автоматизированными. Фактически даже при наличии большого количества автоматизированных тестов всегда остается место для исследовательского тестирования, так как отказаться от ручного тестирования пока просто невозможно.
- Развертывание в рабочем окружении (глава 7, «Развертывание — ввод в эксплуатацию») включает установку приложения в другом окружении, и поэтому данный этап почти не несет никаких рисков. Существуют разные подходы для дальнейшего снижения рисков, связанных с вводом в эксплуатацию.
- В ходе эксплуатации приложения возникают такие проблемы, как необходимость мониторинга и наблюдение за сообщениями, появляющимися в файлах журналов. Эти проблемы обсуждаются в главе 8, «Эксплуатация».

В принципе, выпуск изменений можно прервать на любом из перечисленных этапов. Представьте, что новая версия достигла этапа приемочных испытаний и благополучно преодолела все тесты, но показала слишком низкую производительность в ходе тестирования пропускной способности. В таком случае эта версия не будет передана на следующие этапы, такие как исследовательское тестирование или эксплуатация. Проще говоря, программное обеспечение должно показать, что соответствует все более ужесточающимся требованиям, прежде чем достигнет этапа развертывания в рабочем окружении.

Представим, например, что новая версия содержит логическую ошибку. Такие ошибки должны обнаруживаться самое позднее на этапе приемочных испытаний, где проверяется правильная работа приложения. Как следствие, работа конвейера будет остановлена на этом этапе (рис. 1.4). В этой точке бессмысленно проводить любое дополнительное тестирование.

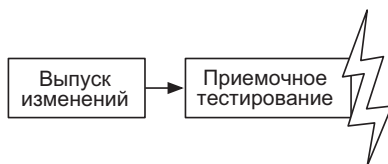


Рис. 1.4. Работа конвейера непрерывного развертывания останавливается на этапе приемочного тестирования

Затем разработчики исправили ошибку и собрали приложение заново. На этот раз оно также успешно преодолело приемочные испытания. Однако в новую функцию закралась еще одна ошибка, не обнаруживаемая автоматизированными приемочными тестами. Эта ошибка может быть вскрыта только в ходе исследовательского тестирования. Как следствие, на этот раз конвейер остановился на этапе исследовательского тестирования, и программное обеспечение не было развернуто в рабочем окружении (рис. 1.5).



Рис. 1.5. Работа конвейера непрерывного развертывания останавливается на этапе исследовательского тестирования

Такой подход помогает не тратить время тестеров понапрасну на проверку программного обеспечения, не соответствующего требованиям или содержащего ошибки, которые обнаруживаются автоматизированными тестами.

В принципе, на одном конвейере могут обрабатываться сразу несколько версий. Конечно, в этом случае конвейер должен поддерживать параллельную обработку нескольких версий — такая обработка невозможна, если тесты выполняются в фиксированных окружениях, потому что окружение будет занято тестированием одной версии и не сможет параллельно тестировать другую версию.

Однако ситуация, когда одновременно обрабатываются две версии, в непрерывном развертывании встречается крайне редко. Проект должен находиться строго в одном состоянии и иметь единственную версию, движущуюся по конвейеру. Самое большее, что может случиться, — изменения в программное обеспечение вносятся так быстро, что новая версия передается в конвейер до того, как предыдущая успеет покинуть его. Исключения могут быть сделаны для исправлений, но главной целью непрерывного развертывания является одинаковая обработка всех версий.

1.5.1. Пример

На протяжении всей книги используется пример приложения, осуществляющего регистрацию клиента в гипотетической компании «Big Money Online Commerce Inc.» (см. раздел П.2). Этот пример специально был максимально упрощен. По сути, приложение регистрирует только имя, фамилию и адрес электронной почты клиента. Сведения, представленные потенциальным клиентом, проверяются. Приложение проверяет синтаксис адреса электронной почты и допускает возможность регистрации только одного клиента для каждого адреса. Кроме того, поддерживается возможность поиска регистрации по адресу электронной почты и ее удаления.

Поскольку приложение не очень сложное, его легко понять, поэтому читатель сможет сосредоточиться на разных аспектах непрерывного развертывания, иллюстрируемых на примере приложения.

Приложение написано на языке Java, с использованием фреймворка Spring Boot, что позволяет запустить приложение с веб-интерфейсом без необходимости устанавливать веб-сервер или сервер приложений. Это упрощает тестирование, так как отпадает потребность в установке какой-либо инфраструктуры. Однако приложение может также выполняться под управлени-

ем сервера приложений или веб-сервера, такого как Apache Tomcat. Данные сохраняются в HSQLDB. Это база данных, хранящаяся в памяти процесса Java. Такая мера, кроме того, снижает техническую сложность приложения.

Исходный код примера можно получить по адресу: <http://github.com/ewolff/user-registration-V2>. Имейте в виду: код примера содержит службы, которые выполняются с привилегиями суперпользователя и доступны из сети. Это неприемлемо для рабочего окружения, потому что может повлечь за собой проблемы безопасности. Однако данный пример предназначался только для экспериментов, в которых требовалось, чтобы пример имел максимально простую структуру.

1.6. В заключение

Перенос программного обеспечения в рабочее окружение — медленный и рискованный процесс. Оптимизация этого процесса может сделать разработку программ более действенной и эффективной. В этом смысле непрерывное развертывание является одним из лучших средств ускорения разработки программных проектов.

Цель непрерывного развертывания — обеспечить ритмичный, воспроизводимый процесс развертывания программного обеспечения во многом подобно тому, как непрерывная интеграция обеспечивает объединение всех изменений. Главным преимуществом непрерывного развертывания является уменьшение сроков ввода в эксплуатацию, однако эта методология способна предложить намного больше: она уменьшает риски, сопутствующие разработке проектов, потому что гарантирует развертывание и запуск программного обеспечения в рабочем окружении. То есть любой проект сможет получить некоторые преимущества — даже если конкуренция не очень велика и сроки ввода в эксплуатацию (вывода на рынок) не играют важной роли.

Ссылки

1. Jez Humble, David Farley: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010, ISBN 978-0-32160-191-9 (*Джез Хамбл, Дейвид Фарли*).
Непрерывное развертывание ПО: автоматизация процессов сборки,

-
- тестирования и внедрения новых версий программ (Signature Series). М.: Вильямс, 2011. ISBN: 978-5-8459-1739-3. — *Примеч. пер.*).
2. <https://martinfowler.com/bliki/ContinuousDelivery.html>
 3. https://en.wikipedia.org/wiki/Continuous_delivery
 4. Eric Ries. Kanban: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Crown Business, 2010, ISBN 978-0-67092-160-7 (*Эрик Рис. Бизнес с нуля. Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели*). М.: Альпина Паблишер, 2016. ISBN: 978-5-9614-6028-5. — *Примеч. пер.*).
 5. David J. Anderson: Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press.
 6. <https://www.sec.gov/litigation/admin/2013/34-70694.pdf>
 7. <http://www.kitchensoap.com/2013/10/29/counterfactuals-knight-capital/>

2

Подготовка инфраструктуры

2.1. Введение

Эта глава посвящена основе непрерывного развертывания: подготовке инфраструктуры. Разные этапы конвейера непрерывного развертывания требуют наличия на компьютере установленного программного обеспечения для выполнения приемочных тестов, тестов пропускной способности или исследовательских тестов. Этот процесс необходимо автоматизировать, так как установка вручную слишком трудоемка. Кроме того, автоматизация существенно снижает риски, связанные с развертыванием новых версий в рабочем окружении, поскольку гарантирует единообразие установки необходимых окружений.

В разделе 2.2 демонстрируется, насколько просто реализовать автоматизацию с использованием сценария установки. Однако применения подобного сценария зачастую бывает недостаточно. Поэтому в разделе 2.3 обсуждается инструмент Chef для автоматизации инфраструктуры. Раздел 2.3.1 описывает технические основы. Chef можно использовать как простой инструмент командной строки (Chef Solo, раздел 2.3.2) или как клиент/серверное решение (разделы 2.3.4 и 2.3.5).

Vagrant (раздел 2.4) — отличный инструмент для установки виртуальных машин на компьютеры разработчиков. Раздел 2.4.1 демонстрирует конкретный пример использования Vagrant и Chef. Раздел 2.4.2 завершает тему обсуждением целесообразности использования Vagrant.

Еще одна возможность — инструмент Docker — описывается в разделе 2.5. Это не только удобная альтернатива виртуализации: Docker использует очень простой подход к установке программного обеспечения в контейнеры. В разделе обсуждается возможность использования Docker совместно с Vagrant (раздел 2.5.4), а также установка Docker на серверы с Docker Machine (раздел 2.5.5), более сложные конфигурации

Docker (раздел 2.5.6) и координация нескольких контейнеров Docker с Docker Compose (раздел 2.5.7). Это простой и доступный способ установки программного обеспечения, при котором конфигурация программных средств на сервере никогда не изменяется — согласно идее «неизменяемого сервера» (раздел 2.6).

Использование таких инструментов, как Chef и Docker, приводит к фундаментальным различиям в процедуре подготовке инфраструктуры. В разделе 2.7 разъясняются особенности технологии Infrastructure as Code («Инфраструктура как код»).

Раздел 2.8 показывает, как для нужд непрерывного развертывания можно использовать облачную технологию Platform as a Service Clouds («Платформа как услуга»).

Наконец, в разделе 2.9 описываются конкретные проблемы, связанные с хранением информации и базами данных. Базы данных особенно сложны в установке и обновлении, потому что хранят большие объемы данных, которые, в зависимости от обстоятельств, приходится переносить в новую версию. Кроме того, совсем непросто сгенерировать и подготовить подходящие тесты для проверки наборов данных приложения. Завершается глава подведением итогов и заключительными замечаниями в разделе 2.10.

2.1.1. Автоматизация инфраструктуры: пример

В разделе П.2 на примере компании «Big Money Online Commerce Inc.», описывался сценарий развертывания новой версии без использования методологии непрерывного развертывания. Однако сотрудники «Big Money Online Commerce Inc.» сделали выводы из случившегося. В качестве одной из мер по предотвращению подобных ситуаций в будущем в компании решили автоматизировать инфраструктуру. Это существенно ускорит создание окружений с необходимым программным обеспечением и сделает процесс более воспроизводимым. Фактически автоматизация инфраструктуры решает следующие проблемы.

- Создание тестового окружения всегда было трудоемкой задачей. Но после автоматизации этой процедуры для создания такого окружения понадобится намного меньше усилий. Это способствует созданию нескольких тестовых окружений, в которых можно провести более полное тестирование.

- В сценарии, представленном в начале книги, ошибка проявилась в рабочем окружении потому, что рабочее окружение отличалось от тестового. Но после автоматизации инфраструктуры она будет воспроизводиться совершенно одинаково. Это верно для обоих окружений — тестового и рабочего. Источник ошибок будет ликвидирован, и они просто станут невозможными.
- Кроме того, легко можно обеспечить ввод в эксплуатацию «ночных выпусков»: это всего лишь автоматизированный процесс, который нужно инициировать. Ошибки, обусловленные выполнением операций вручную, будут исключены. А поскольку процесс развертывания уже используется для создания других окружений, он проверен и надежен.

Для этого группа должна воспользоваться технологиями, способными генерировать сложные окружения. Именно об этом рассказывается в данной главе.

2.2. Сценарии установки

Настройка системы и установка программного обеспечения уже давно автоматизированы: в ПО для Windows используются, например, специальные инсталляторы. Однако для внутреннего ПО, которое требуется развернуть в рабочем окружении, ситуация выглядит иначе: в этом случае настройка и установка часто выполняются вручную. Иногда необходимые шаги описываются в руководстве. Однако даже при использовании контрольных списков для проверки довольно трудно выдержать абсолютно точное следование сложным инструкциям, поэтому этот процесс не застрахован от ошибок и трудно воспроизводим.

2.2.1. Проблемы классических сценариев установки

В некоторых проектах автоматизация установки программного обеспечения реализована с применением сценариев. Сценарий выполняет последовательность шагов, требуемую для создания необходимых файлов и правильных настроек. В этой книге в качестве примера, осуществляющего регистрацию пользователя, необходимо установить сначала сервер Tomcat. В Linux такой сценарий мог бы выглядеть, как показано в листинге 2.1: `apt-get` — это служебная программа, имеющаяся в дистрибутиве Ubuntu

Linux. Сначала командой `apt-get` сценарий обновляет каталог доступных пакетов и устанавливает все доступные обновления. Затем он устанавливает OpenJDK и Tomcat. Наконец, приложение копируется в каталог, где Tomcat ищет выполняемые им приложения. Перед запуском сценария приложение должно быть скопировано на сервер. Также можно организовать загрузку приложения на сервер через HTTP. Это позволит выполнять его установку непосредственно из репозитория.

Листинг 2.1. Сценарий для установки приложения регистрации пользователей

```
#!/bin/sh
apt-get
apt-get dist-upgrade -y
apt-get install -y openjdk-8-jre-headless
apt-get install -y tomcat7
cp /vagrant/user-registration.war /var/lib/tomcat7/webapps/
```

На первый взгляд такой подход выглядит убедительным из-за своей простоты. Однако он не решает многих типичных проблем.

- Он не позволяет выбрать конкретную версию сервера Tomcat для установки или изменить его настройки использования памяти — для этого требуется внести изменения в конфигурационные файлы Tomcat.
- При развертывании новой версии этот сценарий может вызвать проблемы. Приложение в данном примере будет повторно скопировано в каталог сервера Tomcat. Это заставит Tomcat перезапустить приложение, что приведет к его короткой остановке. В сложных приложениях такая остановка может вызвать автоматический выход всех пользователей. Конечно, сценарий можно изменить так, чтобы он копировал только отсутствующие или изменившиеся файлы. Однако это не только усложнит сценарий, но и потребует его дополнительного тестирования.
- Если изменяется конфигурация сервера, этот сценарий не всегда сможет вернуть сервер в предыдущее состояние. Например, если изменяется конфигурация Tomcat, данный сценарий установки не сможет отменить эти изменения.

Помимо перечисленных проблем, которыми «страдает» даже такой простой сценарий, существует еще ряд более серьезных проблем.

- Порой очень непросто установить приложение, включая все необходимые компоненты. А автоматизировать этот процесс еще сложнее. По-

этому из-за слишком высокой стоимости многие попытки никогда не достигают уровня полной автоматизации. В результате до сих пор приходится пользоваться инструкциями и выполнять некоторые операции вручную. Иногда очень трудно выполнить ручные операции в нужном месте — зачастую это требует знания особенностей работы сценария. Инструменты, представленные в этой главе, упрощают реализацию такой автоматизации и сценариев, облегчая достижение уровня полной автоматизации.

- Если установка терпит неудачу, она должна быть перезапущена. В такой ситуации система оказывается в состоянии, когда некоторые части программного обеспечения уже установлены. Простой повторный запуск установки может создать проблемы. Обычно сценарий предполагает, что система находится в состоянии, когда программное обеспечение еще не установлено, поэтому повторная попытка создать папки и файлы может закончиться неудачей и прервать всю процедуру установки. Чтобы устранить подобные проблемы, в сценарий установки добавляется обработка подобных ситуаций, которая тестируется в различных окружениях.

Подобная ситуация может возникнуть и при обновлении установленного программного обеспечения. В этом случае в системе уже имеется установленная версия ПО, требующая обновления. Это означает, что какие-то файлы уже могут присутствовать и должны быть затерты новыми файлами. Сценарий обязан реализовать всю необходимую для этого логику. Кроме того, лишние элементы, ненужные в новой версии, должны быть удалены. Если этого не сделать, могут возникнуть проблемы в работе новой версии, потому что старые настройки и значения могут читаться и использоваться. Однако очень непросто охватить и автоматизировать все варианты обновления, которые могут возникнуть на практике. С другой стороны, эту проблему можно обойти простым созданием сервера «с нуля» для каждой новой версии.

Наконец, часто таких простых сценариев вполне достаточно. Но, когда в дополнение к установке с нуля требуется реализовать внесение изменений в конфигурацию или обновление старой версии, сценарии начинают быстро усложняться. То есть сценарии установки можно использовать, если иметь возможность установки с нуля. Например, Docker (раздел 2.5) хорошо подходит для сборки серверов с самого начала для каждой новой версии программного обеспечения, поэтому этот инструмент часто использует сценарии установки.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

В разделе 2.4 подробно рассказывается о Vagrant, инструменте управления виртуальными машинами. Он поможет вам приобрести опыт создания сценариев установки. Этот инструмент позволяет легко запустить виртуальную машину и сценарий установки.

Чтобы установить Vagrant, выполните следующее.

1. Установите решение виртуализации, например Virtual Box [1].
2. Загрузите и установите Vagrant [2], следуя инструкциям.
3. Установите Git [3].
4. Извлеките репозиторий GitHub командой

```
git clone https://github.com/ewolff/user-registration-V2.git
```

Теперь запустите виртуальную машину в подкаталоге `user-registration/shell` и инициализируйте командой `vagrant up`. В ходе инициализации выполнится сценарий установки.

Сценарий можно также выполнить без запуска машины, командой `vagrant provision`.

- ▲ Доступно ли приложение во время инициализации?
- ▲ Что случится с зарегистрированными пользователями, если повторить инициализацию? Имейте в виду, что приложение использует базу данных, хранящуюся в оперативной памяти, а не на диске.

2.3. Chef

Сценарии установки описывают шаги, необходимые для установки программного обеспечения. Другой подход — описать, как должна выглядеть система после установки программного обеспечения, а не шаги, ведущие к этому состоянию.

Допустим для примера, что для запуска программного обеспечения требуется конфигурационный файл. В классическом подходе просто генерируется новый конфигурационный файл. Однако, если файл уже существует, это может вызвать проблемы: первая — нельзя просто так затереть имеющийся файл. Если сценарий установки затрет прежний файл, это может вызвать перезапуск приложения, включая возможную потерю дан-

ных и приостановку на короткое время. Поэтому замена файла должна быть реализована правильно. Вторая проблема может возникнуть, если неправильно установить права доступа к файлу, то есть если сделать его недоступным для чтения. Все необходимые каталоги также должны существовать. Кроме всего прочего, затирание файла в действующей системе может породить дополнительные проблемы. Например, приложения, обращающиеся к файлу, могут столкнуться с проблемой во время записи в файл.

Альтернативным решением могло бы стать описание содержимого, которое должен иметь файл, и прав доступа к нему. В этом случае в ходе установки можно было бы сравнить желаемое состояние с текущим и соответственно скорректировать файл. Такой подход имеет несколько преимуществ.

- Установку можно повторять так часто, как требуется, — результат всегда должен получаться одинаковым. Такую процедуру установки называют *идемпотентной*¹.
- Обновление до новой версии обычно выполняется довольно просто, потому что не предполагает сборку всей системы заново. Система с установленной старой версией программного обеспечения — особый случай, но и в этой ситуации можно использовать сценарии. Файл может уже существовать, но быть переписанным новой версией, если понадобится.
- Возможность документирования изменений в системе. Каждое изменение в файле или установка определенного программного пакета могут быть отражены в файле журнала. Например, когда записывается файл с новым содержимым, старое содержимое автоматически сохраняется в резервной копии, это позволяет сделать изменения контролируемыми. Опять же сценарий установки должен предусматривать обработку особых случаев.
- Наконец, если файл уже существует и права доступа к нему установлены правильно, его содержимое может не потребовать внесения изменений. Это обстоятельство существенно ускорит процесс установки.
- Часто процесс требуется перезапустить, чтобы быть уверенным, что новый файл действительно прочитан. Перезапуск тоже можно реализовать в сценарии, но только если содержимое файла действительно изменилось.

¹ <https://ru.wikipedia.org/wiki/Идемпотентность>. — Примеч. пер.

В идеале система должна быть описана полностью, включая информацию о версии операционной системы и установленных обновлениях. Это единственный способ гарантировать полную повторяемость установки.

Инструмент Chef, который будет представлен в этом разделе, принадлежит к программным системам, реализующим именно такой подход.

2.3.1. Chef и Puppet

Puppet [4] — еще одна система, использующая подход, аналогичный подходу Chef. В следующей главе будет представлена инфраструктура непрерывной интеграции, сгенерированная с помощью сценария Puppet. Однако ее реализация имеет несколько важных отличий: Puppet использует декларативный подход, предметно-ориентированный язык Ruby DSL (Domain-Specific Language) и структуры данных JSON. Пользователь описывает зависимости, а Puppet устанавливает программное обеспечение в соответствии с планом, созданным на основе списка компонентов и их зависимостей. Chef также поддерживает язык Ruby DSL, но пользователь пишет на нем сценарий установки, а не объявления, как в Puppet. Так как в обоих случаях используется предметно-ориентированный диалект, от пользователя не требуется владеть языком Ruby, чтобы использовать Chef или Puppet. Однако возможности Chef и Puppet легко можно расширить, используя всю мощь Ruby. Кроме того, этот подход проще и понятнее разработчикам.

Другое преимущество обоих подходов — широкая распространенность технологий. Как следствие, существуют сценарии установки для многих типовых программных пакетов, которые вы можете использовать для установки своей системы. Однако будьте внимательны при использовании этих сценариев: в большинстве случаев их нужно адаптировать под конкретные потребности инфраструктуры, потому что, например, в контексте проекта может потребоваться внести некоторые изменения в конфигурационные файлы. Иногда проект может требовать установки дополнительных расширений, не описанных в обобщенном сценарии установки. Также может потребоваться скорректировать некоторые настройки — например, номер сетевого порта, используемого процессом, — не являющиеся частью обычных сценариев. Кроме того, могут поддерживаться не все разновидности операционных систем. Разные дистрибутивы Linux, к примеру, хранят конфигурационные файлы в разных местах, а пакеты программ имеют разные имена. Даже если декларируется, что сценарии учитывают конкретные особенности разных дистрибутивов Linux, в действительности все равно

возникают проблемы, потому что сценарии проверяются не во всех версиях операционной системы.

Кроме различных разновидностей Linux, Puppet и Chef поддерживают также Windows и Mac OS X. Поэтому подходы, описываемые в этом разделе, с успехом используются на этих платформах, однако, как обычно, существуют некоторые различия в деталях. Например, ни Windows, ни Mac OS X не имеют в стандартной конфигурации диспетчера пакета.

Chef и Puppet — открытые проекты и распространяются на условиях лицензии Apache 2.0. Это достаточно свободная лицензия, поэтому нет веских причин не использовать их.

Компании Chef и Puppet Labs выпускают также коммерческие версии. Вы можете пользоваться бесплатными продуктами или приобрести коммерческую поддержку, заплатив за более высокую надежность.

2.3.2. Другие альтернативы

Кроме Puppet и Chef существует множество других инструментов. Рассмотрим некоторые примеры.

- *Ansible* [5] отличается оригинальным синтаксисом сценариев установки, которые называются файлами задач (playbooks). Файлы на языке YAML используются для описания серверов. Серверы устанавливаются удаленно, посредством SSH. Это простой и безопасный подход, он чаще всего не требует наличия в системе дополнительного программного обеспечения. Однако может понадобиться установить библиотеки Python, поскольку инструмент Ansible сам реализован на Python. Кроме Linux он поддерживает также Windows и Mac OS X. Существует очень полезный документ *How to get started with Ansible* («Введение в Ansible») [6]. В нем приводится пример установки Tomcat [7].
- *Salt* — или, точнее, *SaltStack* [8] — также реализован на Python. Инструмент Salt основан на Master Server и Minions, которые выполняются в администрируемых системах как демоны. Эти компоненты взаимодействуют с ZeroMQ — системой обмена сообщениями. Они обеспечивают быстрый и эффективный обмен сообщениями и, соответственно, масштабирование на большое количество систем. Поддерживаются Linux, Mac OS X и Windows. Также имеет обучающее руководство [9].

Развитие Ansible началось в 2012-м, а Salt появился на рынке в 2011-м. То есть эти технологии значительно новее Chef (2009) и Puppet (2005).

Следовательно, они не имеют столь же многочисленного сообщества, и пока еще не накоплен богатый опыт их использования. Однако они решают многие проблемы, которыми «грешили» более ранние технологии.

2.3.3. Технические основы

В общем случае Chef можно использовать тремя разными способами.

- *Chef Solo* — простейший вариант. В этом случае Chef используется как обычный инструмент командной строки — своего рода универсальный сценарий установки. Но при этом на компьютере должна присутствовать вся конфигурация целиком. Это необходимо для создания систем для разработчиков, не имеющих большой инфраструктуры. Данный подход использует, например, Vagrant (раздел 2.4). Таким же способом можно устанавливать серверы. Для этого требуется, чтобы конфигурация извлекалась из системы управления версиями, такой как Git, а затем запускался процесс установки. Такой прием позволяет устанавливать значительные по объему окружения без необходимости иметь центральный сервер. Однако к хранению используемых конфигураций должны применяться повышенные меры безопасности, поскольку они могут содержать пароли к рабочим базам данных. Аналогично должна быть автоматизирована выгрузка изменений на серверы. Это требует повторного запуска Chef Solo на серверах.
- *Chef Server* — настраивает и управляет несколькими клиентами Chef. Сама программа клиента имеет очень маленький размер. Этот подход особенно удобно использовать в тех случаях, когда требуется выполнить установку на несколько компьютеров. Кроме того, он позволяет контролировать все компьютеры с сервера. Следовательно, для каждого компьютера должна быть создана соответствующая запись в системе мониторинга. Кроме того, поддерживается возможность посылать с компьютеров запросы на установку программного обеспечения. Это позволяет, например, автоматически добавить балансировщик нагрузки при установке нового веб-сервера.
- *Chef Zero* — разновидность Chef Server, действующая в оперативной памяти. Благодаря этому она имеет очень маленький размер и высокую скорость работы. Chef Zero особенно удобно использовать для тестирования, когда версия Chef Server оказывается слишком медленной или ее установка чересчур трудоемка.

Имеется также коммерческая версия — *Enterprise Chef*. Она обладает несколькими дополнительными возможностями — например, поддержи-

вает аренду контейнеров или аутентификацию через LDAP или Active Directory. Эта версия доступна также как удаленная система. В этом случае работу сервера Chef Server обеспечивает компания «Chef Inc.». Это очень удобно в тех случаях, когда инфраструктура должна эксплуатироваться в облачном окружении, например в Amazon Web Services. В этом случае можно не беспокоиться об установке Chef Server, резервном копировании или гарантиях доступности. Решение всех этих проблем берет на себя «Chef Inc.».

Основные термины Chef

Основными в настройке систем с помощью Chef являются три термина (рис. 2.1).

- *Ресурсы* (resources) включают все, что может быть настроено или установлено: например, файлы или пакеты Linux. Chef поддерживает большое многообразие ресурсов, таких как репозитории с кодом, сетевые адаптеры и файловые системы.
- *Политики* (policies) определяют, как должны выглядеть ресурсы. Например, политика может требовать наличия учетной записи для пользователя «wolff» или установки пакета «tomcat». Автоматизация с применением Chef в основном заключается в создании подобных политик.
- *Провайдеры* (providers) отвечают за определение текущего состояния ресурса и его изменение в соответствии с требованием политики. Встроенные провайдеры в Chef поддерживают типичные ресурсы операционных систем, как, например, файлы.

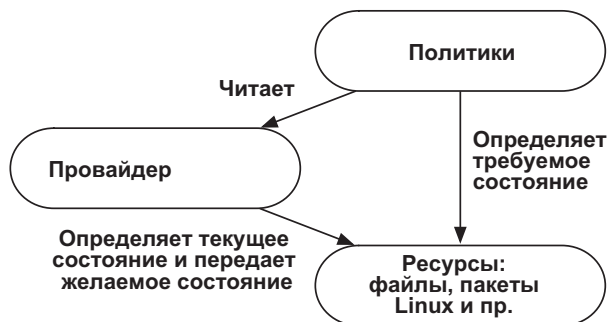


Рис. 2.1. Ресурсы, политики и провайдеры

То есть политики определяют, как должны выглядеть ресурсы, когда Chef закончит работу. Шаги, необходимые для этого, определяются провайдером. Это гарантирует упомянутую *идемпотентность*: независимо от того, как часто будет запускаться Chef, конечный результат всегда один и тот же.

Повара, поваренные книги и рецепты

Предметно-ориентированный язык Chef DSL написан на Ruby. То есть этот язык можно расширять с помощью Ruby. Кроме того, язык Chef позволяет настраивать простые системы без знания языка Ruby. Пользователи даже не замечают, что в действительности они программируют, а не занимаются настройками. Опытные пользователи, в свою очередь, могут расширять системы, используя Ruby. Определяя настройки, пользователи пишут так называемые рецепты (recipes), а это прекрасно сочетается с решением, называемым «Chef» (повар).

Листинг 2.2. Фрагмент рецепта Tomcat

```
template "/etc/tomcat7/server.xml" do
  owner "root"
  group "root"
  mode "0644"
  notifies :restart, resources(:service => "tomcat")
  source "server.xml.erb"
end
```

В листинге 2.2 демонстрируется часть рецепта — в данном случае для Tomcat. Весь проект можно найти по адресу: <https://github.com/ewolff/user-registration-V2>. Прежде всего рецепт определяет пользователя и группу, владеющих файлом, и то, какие права должны устанавливаться для файла. Если файл изменился, служба Tomcat должна быть перезапущена. Эта служба является дополнительным ресурсом Chef, который определяется в другом месте, внутри файла.

Затем следует ссылка на шаблон, который служит основой для создания XML-файла с настройками Tomcat. При использовании шаблонов поддерживается возможность вставки в определенные позиции конкретных значений для определенного сервера, таких как номера сетевых портов. Chef не просто затирает старый файл новым — сначала файл проверяется на соответствие шаблону. И только когда обнаруживаются несовпадения, файл заменяется. Старая версия файла при этом архивируется, это обеспечивает контролируемость изменений. По завершении выполняется проверка при-

надлежащие файлы и права доступа к нему. Служба Tomcat перезапускается, только если произошла замена файла.

Очевидно, что рецепт сам по себе имеет небольшую ценность. Необходимы также дополнительные компоненты, такие как шаблоны. Поэтому рецепт обычно хранится в дереве каталогов: так называемой поваренной книге, или сборнике рецептов (cookbook). Это дерево состоит из каталога с несколькими подкаталогами (рис. 2.2). Обычно сборник рецептов содержит следующие компоненты.

- В каталоге `recipes` находятся обсуждаемые рецепты; они определяют политики для ресурсов. Обычно существует файл рецепта `default.rb`. В сложных случаях имеются дополнительные рецепты, хранящиеся в отдельных файлах.
- Значения внутри рецептов могут зависеть от конкретного сервера. Для этого в рецептах объявляются атрибуты. Все возможные атрибуты и их значения по умолчанию определяются в каталоге `attributes`. Как рассказывается ниже, значения атрибутов могут переопределяться за рамками рецептов, чтобы их можно было использовать в разных контекстах. Таким способом реализуется практика «преимущества соглашений перед конфигурацией». Это означает, что в отсутствие явно заданных в конфигурации значений используются вполне разумные значения по умолчанию.
- В каталоге `files` хранятся дополнительные файлы. Эти файлы копируются на серверы без каких-либо изменений. Однако зачастую практичнее загружать файлы; например, из репозитория.
- Наконец, в каталоге `templates` хранятся все необходимые шаблоны, например шаблоны конфигурационных файлов. Именно в этом каталоге хранится файл `server.xml.erb`, на который ссылается рецепт в листинге 2.2.

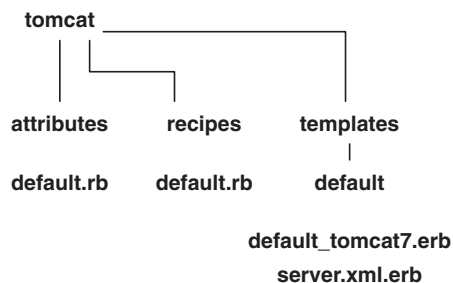


Рис. 2.2. Дерево каталогов рецепта Chef

Для управления сложными сборниками рецептов и зависимостями сборников и рецептов удобно пользоваться Berkshelf [10].

В сборниках рецептов могут также присутствовать другие компоненты, но обычно достаточно перечисленных выше. Давайте вернемся к упомянутому примеру сборника рецептов Tomcat [8]. Листинг 2.2 ссылается на шаблон `server.xml.erb`, который находится в каталоге `templates/default`. Если для компьютеров с определенными именами хостов или версиями операционных систем предполагается использовать другие шаблоны, они должны храниться в подкаталогах с именами, включающими имя хоста или название операционной системы.

В листинге 2.3 демонстрируется фрагмент шаблона `server.xml.erb`. Как можно заметить, в нем встречаются элементы, заключенные в угловые скобки `<%=` и `%>`. Эти элементы служат для вставки значений в шаблоны. В других местах, напротив, используются фиксированные значения. Например, таймаут соединения имеет фиксированное значение 20000. То есть это значение нельзя настроить извне. Чтобы сделать значение настраиваемым, его описание в шаблоне нужно изменить. Это хороший пример ситуации, когда сборник рецептов имеет ограниченную гибкость, легко расширяемую при необходимости. Большинство сборников рецептов размещается на веб-сайтах, таких как GitHub [11]. Эти веб-сайты существенно упрощают возможность изменения компонентов при их интеграции в проекты и обеспечивают совместную разработку сборников рецептов.

Листинг 2.3. Фрагмент шаблона Tomcat

```
...
<Connector port="<%= node["tomcat"]["port"] %>"
  protocol="HTTP/1.1"
  connectionTimeout="20000"
  URIEncoding="UTF-8"
  redirectPort="<%= node["tomcat"]["ssl_port"] %>" />

<Connector executor="tomcatThreadPool"
  port="<%= node["tomcat"]["port"] %>" protocol="HTTP/1.1"
  connectionTimeout="20000"
  redirectPort="<%= node["tomcat"]["ssl_port"] %>" />
...
```

Сборник рецептов также может содержать значения по умолчанию, различающиеся в разных окружениях. С этой целью в подкаталоге `attributes` можно создать файл `default.rb`. В листинге 2.4 показана выдержка из это-

го файла. Как можно заметить, в нем определяются разумные значения по умолчанию.

Листинг 2.4. Выдержка из файла `default.rb`, определяющего значения по умолчанию

```
default["tomcat"]["port"] = 8080
default["tomcat"]["ssl_port"] = 8443
default["tomcat"]["ajp_port"] = 8009
default["tomcat"]["java_options"] = "-Xmx128M -Djava.awt.headless=true"
default["tomcat"]["use_security_manager"] = false

set["tomcat"]["user"] = "tomcat"
set["tomcat"]["group"] = "tomcat"
```

Конечно, можно писать сборники рецептов самостоятельно. Однако не всегда необходимо начинать все с нуля. В Интернете можно найти целые коллекции готовых сборников рецептов [12]. Поэтому всегда имеет смысл сначала поискать подходящие коллекции, прежде чем начинать писать свою. Как уже отмечалось, имеющиеся сборники рецептов часто адаптируются под конкретные потребности проекта. Аналогично, сборники рецептов могут служить примерами и фундаментом для создания собственных сборников.

Если используется сборник рецептов из Интернета, обычно требуется адаптировать его под свой проект, чтобы добиться хороших результатов. Часто необходимо дополнить их поддержкой определенной операционной системы или обеспечить более гибкие возможности настройки в определенных местах — например, введением новых атрибутов. Нередко сборники рецептов для особых случаев оказываются очень простыми и могут быть написаны очень быстро, тогда как уже готовые сборники реализуют обобщенные решения и поэтому оказываются весьма сложными, а кроме того, должны быть расширены под конкретный случай. С этой точки зрения, создание собственного сборника рецептов может оказаться более простой задачей, чем адаптация существующего.

Роли

Чтобы установить сервер, требуется определить рецепты, необходимые для этого. Кроме того, нужно настроить значения атрибутов. Теоретически можно было бы передавать инструменту Chef список рецептов и значения атрибутов при каждой установке. Однако это было бы нелогично. В конце

концов, на каждом этапе конвейера непрерывного развертывания это всегда будет одна и та же разновидность серверов. Кроме того, может потребоваться развернуть несколько идентичных серверов; например, несколько веб-серверов, действующих за балансировщиком нагрузки.

Для этого в Chef существует понятие «ролей»: роль определяет, какие рецепты должны выполняться и какие значения должны присваиваться атрибутам. Для каждого типа серверов требуется определить роль. Впоследствии это упростит установку произвольного количества таких серверов. Помимо этого, Chef Server можно использовать для выполнения запросов, позволяющих узнать, например, сколько серверов играют ту или иную роль и какие это серверы. Таким путем можно установить балансировщик нагрузки, конфигурационные файлы которого содержат все веб-серверы.

Информация о роли в Chef хранится в JSON-файле; как вариант, можно использовать файл с содержимым на языке Ruby DSL. Пример такого файла показан в листинге 2.5. Объявления `json_class` и `chef_type` указывают на то, что определяется роль. Далее следует информация, определяющая сборники рецептов для выполнения и значения атрибутов. В данном случае сначала определяется имя роли — «tomcatserver». Затем следуют определения отдельных атрибутов для сборников рецептов `tomcat` и `webapp`. Наконец, `run_list` определяет рецепты для выполнения.

Листинг 2.5. Конфигурация роли для Chef в формате JSON

```
{
  "json_class": "Chef::Role",
  "chef_type": "role",
  "name": "tomcatserver",
  "description": "Install Tomcat and a web application inside Tomcat",
  "default_attributes": {
    "webapp": {
      "webapp": "demo.war"
    },
    "tomcat": {
      "port": 8080
    }
  },
  "run_list": [
    "recipe[apt]",
    "recipe[webapp]"
  ]
}
```

Каждый сервер — в терминологии Chef его называют узлом (node) — может иметь одну или несколько ролей. В описаниях узлов также можно переопределять атрибуты для индивидуальной настройки каждого сервера.

2.3.4. Chef Solo

Chef Solo предлагает самый простой путь к использованию Chef. В этом случае развертывание программного обеспечения в системе выполняется из командной строки. Это означает, что перед развертыванием в целевую систему должны быть скопированы сценарии установки и настройки. Поэтому логично использовать Chef Solo для тестирования и на локальных компьютерах разработчиков, где эти ограничения не так критичны. В случае распределенной установки на несколько компьютеров также могут использоваться инструменты управления версиями и передачи файлов. Обычно такой подход используется, когда разворачивание инфраструктуры из командной строки оказывается проще установки центрального сервера Chef. Кроме того, данная стратегия помогает исключить центральное узкое место.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Это упражнение поможет читателю получить опыт использования Chef Solo. Ваша задача — установить простое приложение на Java вместе с веб-сервером Tomcat и JDK.

1. Я рекомендую выполнять это упражнение на виртуальной машине. Поэтому установите программное обеспечение, например VirtualBox [13].
2. Затем установите в виртуальную машину Ubuntu 15.04. Образ компакт-диска с дистрибутивом можно получить по адресу: <http://releases.ubuntu.com/15.04/>. Версия сервера имеет важное значение. Обязательно устанавливайте версию 15.04 — в других версиях Ubuntu могут возникнуть проблемы со сценариями.
3. Установите только простой сервер. Не нужно устанавливать специальные пакеты, предлагаемые во время установки. Возможно, проще будет установить сервер SSH и выполнять последующие шаги через `ssh` — в этом случае вы сможете использовать, например, операции копирования/вставки из других окон.

4. Запустите виртуальную машину с установленной операционной системой.
5. Обновите программное обеспечение с помощью команд `sudo apt-get update` и `sudo apt-get upgrade`.

Теперь у вас имеется виртуальная машина с установленной операционной системой Ubuntu без специального программного обеспечения. Следующий шаг — установка Chef.

1. Установите Chef. В ходе установки будет предложено установить сервер — вы можете ответить на него отрицательно. Команда для установки: `sudo apt-get install chef`.
2. Установите систему управления версиями Git командой `sudo apt-get install git-core`.
3. Извлеките содержимое репозитория GitHub командой `git clone https://github.com/ewolff/user-registration-V2.git`.
4. Отредактируйте содержимое файла `solo.rb` в каталоге `chef`: в первой строке определяется переменная `root`. Она должна содержать путь к каталогу с репозиторием Git — например: `root = '/home/ubuntu/user-registration-V2/chef/'`.

Совет: в системе уже имеется предустановленный удобный редактор Nano. Его можно запустить командой `nano`.

1. Теперь, когда все файлы на месте, запустите Chef Solo командой `sudo chef-solo -j node.json -c solo.rb`.
2. После этого должен стать доступным веб-сервер Tomcat. Убедиться в этом можно с помощью команды `curl localhost:8080`, а командой `curl localhost:8080/demo/` можно запустить приложение.

Однако в ответ будет возвращена разметка HTML. В принципе приложение также открывается в браузере, но такое возможно, только если функционирует сетевое соединение с виртуальной машиной.

Наконец, стоит проверить файлы конфигурации `node.json` и `solo.rb`. Кроме того, интересно также заглянуть в роль, используемую Chef, и в разные рецепты.

Для тех, кому интересно, предлагается провести дополнительные эксперименты.

- ▲ Запустите установку повторно — что получилось в результате?

- ▲ Измените номер порта для Tomcat. На выбор можно использовать любой из вариантов: изменить значение по умолчанию в сборнике рецептов или в файле, где определяется роль.
- ▲ Конечно, можно попробовать развернуть приложение в другой версии Ubuntu или Linux. Однако в зависимости от особенностей системы может потребоваться внести в рецепты существенные изменения.
- ▲ Также попробуйте расширить сценарии Chef и таким способом установить дополнительное программное обеспечение. Для этого «позаимствуйте» сборники рецептов из [2] и интегрируйте их в конфигурацию.

2.3.5. Chef Solo: заключение

Это упражнение продемонстрировало выполнение настройки системы с помощью Chef и наглядно показало, насколько трудоемок подобный подход. Прежде всего необходимо установить Chef. Затем требуется откорректировать отдельные файлы, чего в принципе следует избегать при использовании Chef.

Кроме того, теперь необходимо извлечь текущую версию конфигурации из репозитория Git и через равномерные интервалы времени разворачивать изменения; однако никто не использует Chef Solo для сопровождения установленных серверов. Но данный краткий обзор, безусловно, был очень полезен.

2.3.6. Knife и Chef Server

При использовании Chef Server информация о ролях или сборниках рецептов хранится на сервере. Вы тоже можете установить такой сервер Chef [14]. Это необходимо, если требуется сохранить сборники рецептов и роли исключительно в вашем вычислительном центре. В этом случае обязательно должно выполняться резервное копирование ролей и рецептов, необходимых для восстановления после аварии. Однако не должно предъявляться требование к высокой доступности, так как отказ сервера означает, что никакие новые компьютеры не могут быть установлены и роли с рецептами не могут быть изменены.

Альтернативное решение — использовать версию Chef Enterprise, где может использоваться сервер Chef, поддерживаемый компанией «Chef Inc.». В этом случае исключаются дополнительные трудозатраты.

Knife — это «инструмент удаленного управления» для сервера Chef. С его помощью можно управлять ролями, сборниками рецептов и тому подобным на сервере Chef. Но этим круг его возможностей не ограничивается: Knife способен взаимодействовать с решениями виртуализации. Knife может также установить клиента Chef на компьютер. После установки и настройки клиента появится возможность устанавливать программное обеспечение на новый компьютер без ручного вмешательства.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Давайте рассмотрим поближе Chef Server и Knife. Использование облачной инфраструктуры Amazon EC2 и Hosted Enterprise Chef Server позволит без особого труда получить более полное представление о возможностях Chef Server и Knife.

Это упражнение выполняется на любом компьютере. В отличие от предыдущего, оно не требует использования локальной виртуальной машины, однако вы можете воспользоваться ею вновь, чтобы установить программное обеспечение отдельно.

1. Установите Git (см. <http://git-scm.com/book/en/Getting-Started-Installing-Git>).
2. Как и прежде, воспользуемся примером установки сервера Tomcat. Для этого извлеките проект командой `git clone https://github.com/ewolff/user-registration-V2.git`.
3. Затем установите инструмент Knife, входящий в состав Chef Development Kit. Инструкции по установке вы найдете по адресу: <https://downloads.chef.io/chef-dk/>.
4. Наконец, установите плагин Knife EC2 — см. <https://github.com/chef/knife-ec2#installation>.

Теперь у нас установлено все программное обеспечение, необходимое для выполнения упражнения. Следующий шаг — создать учетную запись на Amazon и добавить информацию в конфигурацию.

1. Вам понадобится учетная запись Amazon, которую можно создать на <http://aws.amazon.com/>. Обладателям новой учетной записи

предоставляется большой период бесплатного использования вычислительных ресурсов в облаке. Правда, вы получите в свое распоряжение только микроэкземпляры — наименее мощные виртуальные компьютеры в облаке Amazon Cloud. Однако они обходятся очень дешево, и их мощности обычно достаточно для нужд тестирования.

2. Сгенерируйте ключ доступа по адресу: https://console.aws.amazon.com/iam/home?#security_credential. Он состоит из идентификатора ключа доступа и секретного ключа. Эти ключи нужно присвоить переменным окружения `AWS_ACCESS_KEY_ID` и `AWS_SECRET_ACCESS_KEY`, соответственно, для дальнейшего использования в Knife.
3. Создайте группу безопасности на странице <https://console.aws.amazon.com/ec2/home?region=eu-west-1#s=SecurityGroups>. Она необходима для настройки брандмауэра виртуальной машины. Группа безопасности должна открывать порты 22 (SSH) и 8080/8081 (HTTP с «Дополнительными правилами TCP» — «Custom TCP Rule»). Убедитесь, что ваша группа безопасности не связана ни с каким виртуальным приватным облаком (Virtual Private Cloud, VPC). Введите имя группы безопасности в строке `knife[:groups]` в файле `knife.rb`, который находится в каталоге `user-registration-V2/chef/.chef`.
4. На странице <https://console.aws.amazon.com/ec2/v2/home?region=eu-west-1#KeyPairs> сгенерируйте пару ключей. В результате вы получите файл `.pem`. Сохраните его в каталоге `.chef` и введите имя файла без расширения `.pem` в строке `knife[:aws_ssh_keyname]` в файле `knife.rb`. Установите права доступа к этому файлу командой `chmod 400 *.pem`.

Стоимость услуг Amazon

Запущенные экземпляры потребляют вычислительные ресурсы Amazon, которые стоят денег. Поэтому обязательно зайдите на страницу <https://console.aws.amazon.com/ec2/v2/home?region=eu-west-1>, чтобы увидеть, какие серверы запущены, и завершить их работу, для чего достаточно выбрать действие «terminate» (завершить). Amazon предлагает облачные услуги по всему миру, разделяя его на регионы. Для каждого из регионов определено несколько зон, которые фактически являются отдельными вычислительными центрами. В данном упражнении мы будем использовать регион EU-West-1, соответствующий Европе. Каждый регион отображается в консоли отдельно.

Поэтому важно убедиться, что выбран правильный регион. Впрочем, ссылка выше ведет прямо в нужный нам регион EU-West-1.

Настройкой группы безопасности мы открыли нужные нам порты на машине и в брандмауэре. Ключ SSH будет использоваться инструментом Knife для входа на компьютер и установки Chef. Остальные настройки в `knife.rb` гарантируют, что машина будет действовать в регионе EU-West-1 — в вычислительном центре в Ирландии. Мы будем использовать микроэкземпляр, который не обладает большой вычислительной мощностью и обходится очень недорого. Наконец, использование образа AMI (Amazon Machine Image — образ виртуальной машины Amazon) обеспечит загрузку на экземпляре операционной системы Ubuntu 15.04.

Теперь можно запустить Knife на виртуальной машине с Ubuntu. Удаленный сервер Chef отвечает за хранение и выполнение сборников рецептов. Ниже перечислены шаги по настройке этого сервера.

1. Нам понадобится учетная запись Hosted Enterprise Chef, которую можно получить по адресу: <https://manage.chef.io/>. Учетная запись Hosted Enterprise Chef обслуживается бесплатно до определенного количества компьютеров.
2. Создайте организацию на странице <https://manage.chef.io/organizations/>.
3. Загрузите контрольный ключ для организации и сохраните в каталоге `.chef`.
4. Укажите имя файла с ключом в параметре `validation_key`, в файле `knife.rb`.
5. Укажите имя организации в файле `knife.rb` — определите параметры `validation_client_name` и `chef_server_url`.
6. Щелкните на ссылке Users («Пользователи»). На открывшейся странице создайте ключ для своей учетной записи — файл с расширением `.pem`. Загрузите этот файл, сохраните в каталоге `.chef` и укажите имя файла в строке `client_key` в файле `in knife.rb`.
7. Укажите имя этого файла — без расширения — в параметре `node_name`.

Если теперь, по окончании настроек, просто выполнить в каталоге `.chef` команду `knife node list`, должен появиться вполне пред-

сказуемый ответ, а именно, что пока нет ни одного настроенного сервера.

Следующим шагом будет выгрузка необходимой информации на сервер Chef.

1. Находясь в каталоге `user-registration/chef`, выполните команду `knife cookbook upload -a`. Она выгрузит все сборники рецептов на сервер. Выгрузка веб-приложений может занять некоторое время.
2. Прodelайте то же самое с ролью `tomcatserver`, выполнив команду `knife role from file roles/tomcatserver.json`.
3. Затем запустите команду `knife ec2 server create -r 'role[tomcatserver]' -i .chef/<Amazon PEM>.pem -r 'role[tomcatserver]'`, чтобы запустить новый сервер, куда будут установлены Chef и Tomcat. В аргументе параметра `-i (<Amazon PEM>.pem`, в команде выше) укажите имя файла с парой ключей, полученных на Amazon.

Единственной командой мы запустили сервер, доступный из Интернета. В момент запуска сервера в DNS будет зарегистрировано его общедоступное имя, по которому вы сможете обращаться к нему. Для соединения с приложением воспользуйтесь, например, командой `curl http://<общедоступное_имя_DNS>:8080/demo/`.

Давайте исследуем вновь созданный узел с помощью Knife — это можно сделать командой `knife node list`. Она вернет список всех запущенных узлов. Команда `knife node show <instance-id>` вернет более подробную информацию о конкретном узле `<instance-id>`. Удалить запущенный экземпляр можно командой `knife ec2 server delete <instance-id> --purge`, чтобы избежать неоправданных расходов. Если настройка виртуальной машины была выполнена с ошибками, она может не появиться в списке узлов. В таком случае удалите этот сервер, воспользовавшись консолью EC2.

Еще один важный момент: виртуальные машины, которые будут созданы в ходе упражнения, очень медленные. Машины, создаваемые для эксплуатации, работают намного быстрее. Читателям, желающим поэкспериментировать: существует возможность сгенерировать образы АМІ экземпляров, действующих в консоли EC2, а также экземпляров, запущенных с помощью Knife. В этом случае сохраняется полное содержимое их жесткого диска. Используя такие образы, можно запускать новые АМІ-машины. Они будут содержать полный комплект программ-

ного обеспечения, установленного прежде, это позволит избежать необходимости повторно выполнять процедуру установки и настройки.

Идеи для дополнительных экспериментов.

- ▲ Попробуйте следить за работой выполняющихся узлов в консоли AWS по адресу: <https://console.aws.amazon.com/>, в панели управления EC2 региона EU-West-1.
- ▲ Попробуйте установить и запустить программное обеспечение в другой версии Ubuntu. Имена образов AMI для ввода в `knife.rb` можно найти по адресу: <http://uecimages.ubuntu.com/>.
- ▲ Попробуйте запускать машины в других регионах. Для этого используйте другой образ — список имеющихся образов находится по адресу: <http://uec-images.ubuntu.com/vivid/current/>. Выбранный образ и имя региона следует ввести в файл `knife.rb`. Кроме того, для каждого региона на сайте Amazon следует сгенерировать свою пару ключей SSH.

2.3.7. Chef Server: заключение

Решение на основе Knife и Chef Server идеально подходит для установки сложных окружений, включающих несколько серверов. Новый сервер можно запустить единственной командой. Кроме того, существующие типы серверов, рецепты и роли контролируются в фоновом режиме. То есть весь комплекс машин можно сгенерировать с помощью сервера Chef, необходимых конфигураций Chef и Knife. Этот подход позволяет также быстро создавать окружения для тестирования программного обеспечения.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Главная цель этой главы — познакомить вас с основными технологиями, основанными на сценариях. Знакомство с этими темами можно продолжить в следующих направлениях.

- ▲ Альтернативой инструменту Chef, представленному выше, может служить Puppet. Для знакомства с этим инструментом по адресу <https://puppetlabs.com/download-learning-vm> вы получите виртуальную машину.

- ▲ Разумеется, важно также продолжить изучение Chef — исчерпывающая информация находится по адресу: <https://learn.chef.io/>.
- ▲ Наконец, имеет смысл заняться глубоким изучением рецептов Chef [15]. Как автоматизировать развертывание существующего приложения с использованием этих инструментов? Каких сборников рецептов не хватает? Обладают ли сборники достаточной гибкостью?

2.4. Vagrant

Чтобы использовать Chef, необходимо иметь подходящий компьютер с установленной минимальной операционной системой и клиентом Chef. Именно это предлагает Knife (см. раздел 2.3.4); однако затем требуется установить Chef Server или Hosted Chef. Кроме того, должна быть подготовлена соответствующая облачная инфраструктура или инфраструктура виртуализации.

Такой подход оказывается слишком трудоемким в случаях, когда разработчикам просто необходимо что-то быстро протестировать. Для таких ситуаций идеально подходит Vagrant [16]: для создания тестового окружения используется виртуализация. Vagrant поддерживает решение Virtual Box [17], которое распространяется на условиях открытой лицензии GPL и может использоваться бесплатно. Кроме того, коммерческая версия Vagrant поддерживает VMware Workstation и VMware Fusion в Windows/Linux и Mac OS X соответственно. Vagrant имеет модульную организацию и расширяется при помощи плагинов [18], среди которых имеется несколько плагинов для использования других решений виртуализации. Благодаря этому Vagrant можно расширить и превратить в универсальный инструмент «удаленного управления» виртуальными машинами, подобный инструменту Knife (раздел 2.3.4). В отличие от Knife, Vagrant не ограничивается применением Chef для развертывания программного обеспечения.

Помимо операционной системы виртуальная машина содержит соответствующие компоненты Chef или Puppet, необходимые для установки дополнительного программного обеспечения. В ней также установлен сервер SSH с настроенным режимом аутентификации посредством сертификатов, известных Vagrant. Это дает инструменту Vagrant возможность подключаться к виртуальной машине и выполнять необходимые действия. Кроме того, существует возможность внедрять в файловую систему вир-

туальной машины фрагменты файловой системы носителя. Как уже упоминалось, для этого требуется установить расширения на виртуальной машине. Поскольку полная конфигурация слишком сложна, не имеет смысла заниматься ею самостоятельно. Используйте одну из готовых виртуальных машин [19] или создавайте такие машины с помощью *veewee* [20] или *Packer* [21]. *Packer* — более современное решение и обладает более развитыми возможностями. Эти инструменты поддерживают разные операционные системы для установки на виртуальную машину — не только разные дистрибутивы Linux, но также Windows.

Установка Vagrant позволяет выполнить следующие действия (рис. 2.3).

- Импортировать пустую виртуальную машину.
- Настроить виртуальную машину так, чтобы она могла использовать файлы в системе-носителе. Таким способом виртуальная машина, например, получит доступ к рецептам Chef.
- После этого Vagrant сможет подключиться к виртуальной машине через SSH и установить все необходимое с помощью Chef Solo (раздел 2.3.2). Кроме Chef Solo можно также использовать Puppet или простейшие сценарии установки на языке командной оболочки.

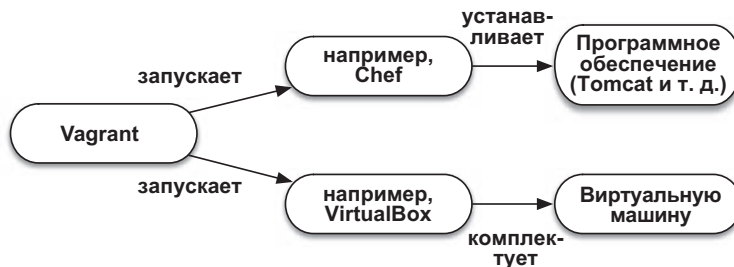


Рис. 2.3. Vagrant запускает виртуальную машину с использованием, например, Virtualbox и устанавливает на нее необходимое программное обеспечение, например Chef

Vagrant можно также использовать для создания и комплектации нескольких виртуальных машин.

Когда предпочтительнее использовать Vagrant? Vagrant легко установить на ноутбук или на компьютер разработчика. Дополнительно необходимо установить программное обеспечение для виртуализации. При наличии

этих двух составляющих легко автоматически установить на локальном компьютере одну или несколько виртуальных машин. Поскольку те же инструменты можно использовать для развертывания рабочего окружения, разработчики получают возможность опробовать разрабатываемое программное обеспечение в окружении, близком к рабочему.

2.4.1. Пример с Chef и Vagrant

Чтобы полнее проиллюстрировать возможности Vagrant, давайте вернемся к примеру из раздела 2.3.2. Мы снова попробуем установить на компьютер Java, сервер Tomcat и веб-приложение на Java.

Центральным компонентом Vagrant является конфигурационный файл Vagrantfile. Фактически этот файл содержит программный код на языке Ruby, при этом он может не выглядеть таковым на первый взгляд. В действительности здесь используется предметно-ориентированный язык Ruby DSL, подобный языку Chef, который мы видели выше.

Листинг 2.6. Vagrantfile

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/vivid64"

  config.vm.network "forwarded_port", guest: 8080, host: 18080
  config.vm.network "forwarded_port", guest: 8081, host: 18081

  config.vm.provision :chef_solo do |chef|
    chef.cookbooks_path = ["cookbooks"]
    chef.roles_path=["roles"]
    chef.add_role("tomcatserver")
  end
end
```

В листинге 2.6 демонстрируется пример содержимого файла Vagrantfile. Здесь определяется, какой образ должен использоваться для установки системы. Этот образ хранится локально. Далее определяются параметры перенаправления сетевых портов: некоторые порты компьютера перенаправляются в порты виртуальной машины. Если, например, попробовать обратиться к URL <http://localhost:18080/>, запрос будет перенаправлен в порт 8080 виртуальной машины, где должен работать Tomcat. В конце следуют настройки развертывания программного обеспечения с помощью Chef. В них определяются пути к сборникам рецептов и роли, включая роль, присваиваемую серверу. Поскольку виртуальная машина имеет до-

ступ к каталогам системы-носителя, через них легко можно передать в виртуальную машину сборники рецептов и роли.

Запускается система командой `vagrant up`. Она требует, чтобы файл `Vagrantfile` находился в текущем каталоге. Команда загрузит указанный в настройках образ и установит программное обеспечение. Получить список других поддерживаемых команд можно командой `vagrant help`. Команда `vagrant provision`, например, используется для запуска Chef. То есть она не нужна для сборки совершенно новой виртуальной машины — команда `vagrant provision` необходима только для обновления и может сэкономить немало времени. Командой `vagrant halt` можно остановить работу окружения — она просто завершит выполнение виртуальной машины. Команда `vagrant destroy`, с другой стороны, полностью уничтожит виртуальную машину, включая все файлы, хранящиеся в ней. Наконец, команда `vagrant ssh` открывает доступ к командной оболочке в виртуальной машине.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Для начала установим Vagrant.

1. Сначала установите программное обеспечение для виртуализации, например Virtual Box [22].
2. Загрузите Vagrant [23] и установите, следуя инструкциям.
3. Скопируйте репозиторий GitHub командой `git clone https://github.com/ewolff/user-registration-V2.git`.
4. После этого создайте и разверните виртуальную машину в подкаталоге `user-registration/chef` командой `vagrant up`.

Дальнейшие эксперименты можно выполнять в разных направлениях.

- ▲ Выше уже были перечислены некоторые команды Vagrant. Если попробовать запустить Vagrant без команды, программа выведет список поддерживаемых команд. Опробуйте их.
- ▲ `vagrant-cachier` [24] позволяет кэшировать пакеты Linux для установки — это увеличивает скорость создания виртуальных машин с помощью Vagrant, так как отпадает необходимость повторной загрузки больших объемов данных из Интернета. Измените `Vagrantfile` для поддержки этой возможности.
- ▲ Попробуйте изменить сетевые порты для Tomcat. Для этого необходимо внести изменения в роль. Порт 8081 — хороший выбор,

потому что его перенаправление уже настроено в файле конфигурации Vagrantfile.

- ▲ Список плагинов [25] для Vagrant весьма обширен. Вам будет полезно ознакомиться с ним, чтобы получить представление об имеющихся возможностях, а также опробовать некоторые плагины по вашему усмотрению.
- ▲ Кроме того, для сборки собственной виртуальной машины можно использовать инструмент Packer [26], поддерживающий интеграцию с Vagrant.
- ▲ Vagrant также позволяет создать несколько виртуальных машин: см. <http://docs.vagrantup.com/v2/multi-machine/>. Почему бы не попробовать создать дополнительный сервер Tomcat или сервер баз данных? С помощью vagrant-hostmanager [27] различные виртуальные машины в Vagrant смогут находить друг друга по именам.

2.4.2. Vagrant: заключение

Как мы видим, Vagrant отлично подходит для установки серверов и обеспечения идентичности окружений на компьютерах разработчиков. В отличие от установки Virtual Box и подготовки виртуальных машин вручную, этот инструмент существенно облегчает труд. Плагины для Vagrant обеспечивают еще более интересные возможности: благодаря им Vagrant можно рассматривать как универсальный инструмент для установки и настройки виртуальных машин. Поэтому Vagrant можно считать обязательным инструментом для разработчиков, нуждающихся в автоматизации инфраструктуры.

2.5. Docker

Непрерывное развертывание основано на виртуальных машинах, потому что практически невозможно использовать физические машины для поддержки разных этапов конвейера непрерывного развертывания. Виртуальные машины полностью отделены друг от друга — каждая имеет собственные виртуальные жесткие диски и собственную операционную систему. Однако, чтобы свести к минимуму усилия по подготовке окружений для приложения, обычно на всех машинах устанавливается одна и та же операционная система с одинаковым набором других компонентов окружений.

Приложение в большинстве случаев требует наличия нескольких машин, например сервера баз данных и сервера приложений.

Такой подход влечет распыление ресурсов: виртуальные машины в значительной степени идентичны, но настраиваются абсолютно независимо друг от друга. Было бы лучше, если бы общие их части хранились в единственном экземпляре.

Увеличение эффективности виртуализации также выгодно в некоторых других отношениях: даже при том что такие инструменты, как Chef, позволяют укомплектовать виртуальные машины необходимым программным обеспечением, такой подход влечет значительные расходы — установка должна выполняться не только на новые машины, но также на машины, где программное обеспечение уже установлено. Однако, если бы было возможно быстро подготовить и запустить новую виртуальную машину, старую виртуальную машину со старым программным обеспечением можно было бы просто удалить и пользоваться новой. Это значительно упростило бы установку программного обеспечения. В этом контексте интересно отметить серверы Phoenix [28]: серверы, которые могут создаваться в любой момент, подобно сказочной птице Феникс, возникающей из пепла.

2.5.1. Решение на основе Docker

Docker [29] предлагает намного более эффективное решение виртуализации. Высокая эффективность достигается за счет использования нескольких разных технологий.

- Вместо полной виртуализации в Docker используются так называемые контейнеры Linux (Linux Containers, LXC), которые в свою очередь опираются на механизм cgroups в ядре Linux. Это позволяет реализовать легковесную альтернативу виртуальным машинам: все контейнеры используют одну и ту же базовую операционную систему. Поэтому в памяти находится только одно ядро. Однако процессы, сетевая и файловая системы и каталоги пользователей хранятся отдельно. В сравнении с виртуальной машиной (ВМ) контейнер имеет намного более низкие накладные расходы — на простом ноутбуке легко можно запустить несколько сотен контейнеров. Кроме того, подготовка и запуск нового контейнера занимают меньше времени, чем подготовка и запуск виртуальной машины, поскольку не требуется тратить время на загрузку операционной системы — достаточно запустить новый процесс. Контейнер проще в управлении, потому что требует только индивидуальной настройки ресурсов операционной системы.

Предполагается, что в будущем помимо базовой технологии LXC Docker будет поддерживать также BSD Jails и Solaris Zones.

- Файловая система также оптимизирована: имеется возможность использовать базовые образы, доступные только для чтения. В то же время в контейнеры могут внедряться дополнительные файловые системы, доступные для записи. Одна файловая система может накладываться на другую. Это позволяет, например, генерировать базовые образы с операционной системой. Если в действующий контейнер устанавливается новое программное обеспечение или изменяется содержимое файлов, сохранять требуется только измененные данные. Это значительно снижает требования к доступному пространству на жестком диске.
- Контейнеры обладают интересными дополнительными возможностями: например, базовый образ может содержать только операционную систему, куда впоследствии устанавливается программное обеспечение. Как упоминалось выше, сохранить в файловой системе потребуется только изменения, произведенные в ходе установки. На основе этой разности можно сгенерировать дополнительный образ. Затем – запустить контейнер, добавляющий этот дополнительный образ к базовому, с операционной системой. Разумеется, впоследствии в такой контейнер можно установить дополнительное программное обеспечение и создать еще один дополнительный образ. При таком подходе каждый новый «слой» в файловой системе будет содержать только отличия от предыдущего. Во время выполнения фактическая файловая система может компоноваться из подобных слоев, что обеспечивает эффективное повторное использование установленного программного обеспечения.
- На рис. 2.4 показан пример файловой системы действующего контейнера. На самом нижнем уровне находится операционная система Ubuntu.



Рис. 2.4. Файловая система в Docker

Поверх него располагается Java. Затем следует пример приложения. Чтобы контейнер мог сохранять изменения, на самом верху находится дополнительная файловая система, куда записываются измененные файлы. Если контейнеру потребуется прочитать файл, он будет выполнять поиск требуемого файла на всех уровнях, начиная с верхнего, пока не найдет его.

Контейнеры Docker и виртуализация

Итак, контейнеры Docker предлагают более эффективную альтернативу виртуализации. Однако это не настоящая виртуализация, потому что контейнеру принадлежит лишь часть ресурсов — он имеет свою память и собственную файловую систему, — но все контейнеры вместе используют, например, общее ядро. Соответственно, этот подход имеет определенные недостатки. Контейнер Docker может использовать только Linux и только то ядро, что установлено в системе-носителе, — то есть в контейнере нельзя, например, запустить приложение для Windows. Кроме того, границы между контейнерами не столь же непреодолимые, как между виртуальными машинами. Ошибка в ядре повлияет на работу всех контейнеров Docker. Более того, Docker не работает в Mac OS X и Windows — однако эту проблему можно преодолеть с помощью Vagrant (раздел 2.5.4) или Docker Machine (раздел 2.5.5). На этих платформах можно установить Docker непосредственно, а «за сценой» контейнеры будут выполняться виртуальной Linux-машиной.

Цели Docker

Основная цель Docker — использовать контейнеры для распределения программного обеспечения. В сравнении с обычными процессами Linux контейнеры намного лучше изолируют отдельные приложения. Кроме того, когда потребуется установить программное обеспечение в другой системе, можно просто перенести соответствующий образ файловой системы. Docker поддерживает репозиторий образов, благодаря которому можно запустить большое количество серверов с идентичными образами.

Взаимодействия между контейнерами Docker

Контейнерам Docker необходимо так или иначе взаимодействовать друг с другом. Например, приложение на веб-сервере должно иметь возможность взаимодействовать с базой данных. С этой целью контейнеры ис-

пользуют сетевые порты. Кроме того, можно использовать общие разделы файловой системы: один контейнер может записывать данные, а другой — читать.

В результате получается компонентная модель, в которой программное обеспечение распределено по отдельным контейнерам Docker и взаимодействует друг с другом посредством сетевых портов и файловой системы. Это обеспечивает надежное разделение программного обеспечения. В целом контейнеры позволяют получить более дробную организацию, чем типичные виртуальные машины. В разделе 8.3 мы добавим анализ файлов журналов примера приложения, а для этого воспользуемся контейнерами Docker. При использовании истинной виртуализации все эти компоненты, скорее всего, пришлось бы запускать на одной виртуальной машине, чтобы не расходовать ресурсы понапрасну.

2.5.2. Создание контейнеров Docker

В принципе, контейнеры Docker можно устанавливать с использованием таких решений, как Chef или Puppet. Главное преимущество этих инструментов — воспроизводимость операций по установке и возможность строго определить, как должна выглядеть система после установки. Это значительно упрощает обновления — сценарии легко приспособить под разные требования, изменяя параметры, благодаря чему их можно использовать повторно. Однако разработка таких сценариев требует сил и времени.

С использованием Docker задача обновления окружения теряет свою важность: запуск нового контейнера требует намного меньше накладных расходов, чем запуск виртуальной машины. То же верно в отношении установки контейнеров. Кроме того, контейнеры Docker позволяют получить более дробную организацию. Поэтому в каждый отдельный контейнер требуется устанавливать меньше программного обеспечения. Это делает процедуру установки проще и требует меньше ресурсов.

В дополнение к удобству установки с применением Chef или Puppet контейнеры Docker выглядят привлекательной альтернативой еще и тем, что позволяют получить совершенно новый контейнер, когда требуется развернуть новую версию программы или компонента. Создание совершенно нового образа требует не так много работы. Это существенно снижает трудозатраты на автоматизацию установки, так как начальной точкой может служить совершенно новый контейнер. Как следствие, отпадает необходимость обновлять старую версию системы.

Если использовать Puppet или Chef в комбинации с Docker нежелательно, можно с помощью Packer (см. ссылку [26] в конце главы) сгенерировать подходящие образы для Docker, уже содержащие Chef или Puppet.

Файлы Dockerfile

Образы Docker можно создавать в интерактивном режиме. В этом случае пользователь просто запускает контейнер и устанавливает необходимое программное обеспечение, выполняя последовательности команд. В конце создается образ сгенерированного контейнера, чтобы потом можно было создать сколь угодно много экземпляров контейнера в точно заданном состоянии. Однако с выходом новой версии программного обеспечения весь процесс установки придется пройти заново. Альтернативное решение предлагают файлы Dockerfile. Они позволяют автоматизировать создание образов, которые могут служить основой файловой системы для контейнера Docker. Пишутся файлы Dockerfile довольно просто.

Пример такого файла показан в листинге 2.7. Этот файл описывает установку Java в образ для контейнеров Docker. Этот образ послужит основой для установки дополнительного программного обеспечения, такого как веб-сервер Tomcat или приложения на Java.

Листинг 2.7. Файл Dockerfile для установки Java

```
FROM ubuntu:15.04
RUN apt-get update; apt-get dist-upgrade -y
RUN apt-get install -y openjdk-8-jre-headless
```

Директива FROM импортирует указанный в ней образ. В данном случае используется начальная установка Ubuntu. Перед двоеточием стоит имя образа, а после двоеточия — тег, в данном случае номер версии Ubuntu. Этот образ загружается из общедоступного репозитория Docker. Подобные образы значительно меньше типовой установки Ubuntu. В образ не включены многие сетевые службы и привычные инструменты ради максимального сокращения его размера. Собственно установка осуществляется директивами RUN. Они выполняются в контейнере. Первая директива RUN обновляет установленную версию Ubuntu. В зависимости от момента времени, когда запущен сценарий, будут применены разные обновления, то есть, строго говоря, данная установка не воспроизводима. Поэтому в рабочем окружении имеет смысл начинать с фиксированного образа, такого как Ubuntu 15.04, используемого в директиве FROM в данном примере конфигурационного файла. Эта установка является полностью воспроизводимой. Вторая директива RUN устанавливает в образ поддержку Java.

«За кулисами» результат выполнения каждой строки в `Dockerfile` фиксируется отдельно: изменения в файловой системе контейнера записываются в отдельный слой внутри файловой системы (как показано на рис. 2.4). Соответственно, в результате выполнения этого файла `Dockerfile` появятся два образа:

- образ с обновленной установкой Ubuntu;
- образ с установкой Java.

Когда другой `Dockerfile` также обновит установку Ubuntu точно такой же директивой `RUN`, это не приведет к созданию дополнительного образа, потому что будет использоваться уже имеющийся промежуточный образ. В этой области Docker оптимизирован для достижения максимальной эффективности.

Наконец, Docker поддерживает очень простой подход для создания образов и контейнеров, основанный исключительно на применении простых сценариев.

Сборка и запуск образов Docker

Чтобы создать образ Docker, необходимо использовать инструмент командной строки `docker`. Например, сгенерировать образ с установленной поддержкой Java можно командой

```
docker build -t java java
```

Docker отыщет в подкаталоге `java` файл `Dockerfile` и создаст образ, опираясь на него. Он отметит его тегом `java`, как указано в параметре `-t`. Создание образа может потребовать некоторого времени для загрузки начального образа Ubuntu и различных пакетов.

Если понадобится запустить новый контейнер на основе этого образа, это можно сделать командой

```
docker run java echo "hello"
```

Эта команда запустит контейнер на основе сгенерированного образа и выполнит в нем команду `echo "hello"`. После этого контейнер автоматически завершит работу.

Команда `docker run -i -t java /bin/bash` запустит контейнер на основе сгенерированного образа, но на этот раз с командной оболочкой внутри, ожида-

ющей ввода команд. Параметр `-i` предотвращает немедленное завершение контейнера, а параметр `-t` подключает виртуальный терминал к контейнеру. Поэтому любые последующие команды будут немедленно передаваться в контейнер.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Далее в этой главе рассматриваются дополнительные подробности о Docker, а глава 8, «Эксплуатация», демонстрирует использование этого инструмента на примере создания сложного окружения. Однако уже сейчас можно немного попрактиковаться.

- ▲ Найдите электронную документацию с описанием Docker.
- ▲ Опираясь на информацию в этой главе, создайте образ Docker с Java. Найдите образы, созданные в дополнение к базовому образу с Ubuntu, и образ с установленной поддержкой Java. Подсказка: если выполнить команду `docker` без параметров, она выведет список всех возможных команд и среди них команду, позволяющую получить список образов, если вызвать ее с соответствующими параметрами. Если вызвать команду `docker` с параметром `-h`, она перечислит все поддерживаемые параметры.

2.5.3. Запуск примера приложения с помощью Docker

Образы Docker из раздела 2.5.2 можно использовать для запуска примера приложения в контейнере Docker. Необходимый для этого файл `Dockerfile` показан в листинге 2.8. Директива `COPY` копирует приложение из файловой системы в образ Docker. Директива `CMD` определяет команду для выполнения после запуска контейнера. Эта команда запускает приложение. Файл журнала приложения сохраняется в каталоге `/log`. Директива `EXPOSE` открывает порт 8080 этого контейнера для доступа извне.

Листинг 2.8. Файл `Dockerfile` для запуска примера приложения

```
FROM java
COPY user-registration-application-0.0.1-SNAPSHOT.war user-registration-
  application-0.0.1-SNAPSHOT.war
CMD /usr/bin/java -Dlogging.path=/log/ -jar user-registration-application-0.0.1-
  SNAPSHOT.war
EXPOSE 8080
```

Создание образа осуществляется в точности так, как описывалось в предыдущем разделе. Запуск образа выглядит следующим образом:

```
docker run -p 8080:8080 -v /log:/log user-registration
```

Параметр `-p` подключает порт 8080 контейнера к порту 8080 системы-носителя, чтобы приложение было доступно по этому порту системе-носителю. Аналогично, каталог `/log` системы-носителя связывается с каталогом `/log` контейнера. Таким способом контейнеру предоставляется доступ к каталогам системы-носителя.

Для нужд тестирования имеет смысл не сохранять выполняемый файл внутри образа Docker, а читать его из файловой системы носителя. В этом случае любое изменение в приложении будет сразу же доступно в контейнере Docker. Достаточно лишь смонтировать каталог `«/target»` контейнера в каталог сборки и изменить команду запуска процесса Java так, чтобы она извлекала выполняемый файл программы из этого каталога.

В идеале результат сборки должен загружаться из репозитория на сервере и затем копироваться в образ. Также возможно генерировать образ Docker непосредственно в процессе сборки.

Дополнительные команды Docker

Команда `docker ps` показывает работающие в настоящий момент контейнеры, а команда `docker ps -a` дополнительно показывает остановленные контейнеры. Команда `docker logs` показывает записи в журнале контейнера — в качестве параметра команде должен передаваться идентификационный номер контейнера. Образы могут сохраняться и загружаться из репозитория — эти операции выполняются командами `docker push` и `docker pull`.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Скопируйте к себе проект примера [30] из репозитория с помощью инструмента командной строки Git [31]:

```
git clone https://github.com/ewolff/user-registration-V2.git
```

Описание установки приложения можно найти в подкаталоге `docker` (<https://docs.docker.com/installation/>).

1. Сначала скомпилируйте приложение командой `mvn install` в подкаталоге `user-registration-application`.

2. Затем сгенерируйте в подкаталоге `docker` образы Docker с Java и приложением регистрации пользователя. Это можно сделать следующими командами: `docker build -t java java`. Не забывайте, что вы должны создать два образа.
3. Выполните команду `docker run -p 8080:8080 -v /log:/log user-registration` для запуска приложения в контейнере Docker. Возможно, вам придется использовать другой каталог вместо `/log` для сохранения файлов журналов. Этот каталог будет содержать вывод процессов в виде файлов журналов, что упростит поиск возможных ошибок.
4. Сгенерируйте образ Docker как часть процесса сборки, а не отдельной командой, вызывающей Docker. Существуют плагины для Maven, позволяющие напрямую генерировать образы Docker. Такой плагин можно интегрировать в процедуру сборки, чтобы генерировать образ Docker в ее ходе.

2.5.4. Docker и Vagrant

Vagrant (см. раздел 2.4) может использовать Docker в роли провайдера. В этом случае создаются не виртуальные машины в VirtualBox, а контейнеры Docker в Linux. Это позволяет легко сгенерировать на основе образов систему, состоящую из нескольких контейнеров Docker, и запустить контейнеры с требуемыми параметрами, объявленными в файле Vagrantfile. Кроме того, упрощается доступ к файловой системе носителя.

Однако подход с применением Docker существенно отличается от подхода на основе VirtualBox: инструменты Chef и Puppet редко бывают доступны в образах Docker. Кроме того, не всегда существует возможность соединиться с машиной по SSH из-за отсутствия в контейнере сервера SSH. Поэтому недостаточно просто заменить VirtualBox на Docker.

Комплектация контейнеров с помощью Vagrant

Альтернативой инструментам Chef и Puppet могут служить файлы Dockerfile. То есть конфигурация контейнеров определяется с помощью файлов Dockerfile вместо Chef или Puppet. В этом случае Vagrant запустит виртуальную машину, установит туда Docker и с помощью файлов Dockerfile сгенерирует и запустит необходимые контейнеры.

Листинг 2.9. Файл конфигурации Vagrant для развертывания примера приложения

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/vivid64"
  config.vm.synced_folder "log", "/log", create: true
  config.vm.network "forwarded_port", guest: 8080, host: 8090

  config.vm.provision "docker" do |d|
    d.build_image "--tag=java /vagrant/java"
    d.build_image "--tag=user-registration /vagrant/user-registration"
  end

  config.vm.provision "docker", run: "always" do |d|
    d.run "user-registration",
      args: "-p 8080:8080 -v /log:/log"
  end
end
```

Для развертывания примера приложения в контейнере с применением файлов Dockerfile, представленных в разделах 2.5.3 и 2.5.2, можно использовать файл конфигурации Vagrant (Vagrantfile) из листинга 2.9. Основой служит виртуальная машина с Ubuntu, куда Vagrant устанавливает Docker. Строки `d.build_image` генерируют соответствующие образы. Это происходит в процессе подготовки виртуальной машины, то есть в первом вызове команды `vagrant up`. Параметр `run: "always"` гарантирует запуск контейнера Docker, описанного в строке `d.run`, в каждом (не только в первом) вызове команды `vagrant up`.

Остальная часть конфигурации описывает подключение каталогов и сетевых портов системы-носителя, в которой действует Vagrant, к каталогам и портам виртуальной машины, где действует Docker, и, соответственно, к каталогам и портам контейнера Docker.

На рис. 2.5 изображены три разных уровня.

- Конфигурационный файл Vagrantfile связывает каталог `log` в системе-носителе с каталогом `/log` в виртуальной машине Vagrant (параметр `config.vm.synced_folder`). А параметр `-v` в строке `d.run` отображает этот каталог в каталог `/log` контейнера Docker.
- Порт 8090 системы-носителя отображается в порт 8080 в виртуальной машине Vagrant и в порт 8080 контейнера Docker. То есть приложение будет доступно по адресу URL `http://localhost:8090`.
- Команда `vagrant up` запустит виртуальную машину Vagrant, сгенерирует образы Docker и запустит контейнеры. Команда `vagrant provision` только сгенерирует образы Docker и запустит контейнеры. Кроме того,

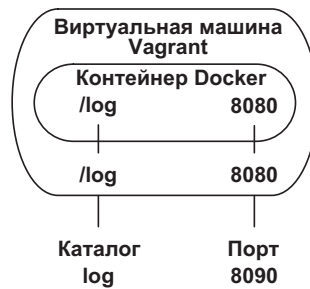


Рис. 2.5. Привязка каталогов и портов в цепочке от системы-носителя через Vagrant к Docker

можно воспользоваться инструментом командной строки Docker. Однако для этого в виртуальной машине Vagrant должен быть запущен сеанс командой `vagrant ssh`.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

1. Установите Vagrant (раздел 2.4.1).
2. Скопируйте проект примера [32]. При использовании инструмента командной строки Git [33] это можно сделать командой `git clone https://github.com/ewolff/user-registration-V2.git`.
3. В проекте примера, в каталоге `docker`, имеется готовый файл `vagrantfile`. Он разворачивает пример приложения в контейнере Docker с помощью Vagrant. Запустите это окружение.

Это решение скопирует приложение в образ Docker. С другой стороны, с помощью Maven приложение можно собрать непосредственно в системе-носителе, а затем смонтировать каталог с приложением в образ Docker. Для этого сначала нужно смонтировать подкаталог `target` в системе-носителе в виртуальную машину Vagrant, добавив строку `config.vm.synced_folder` в конфигурационный файл `Vagrantfile`. Затем каталог в виртуальной машине нужно смонтировать в контейнер Docker. В файле `Vagrantfile` имеется пример монтирования каталога `log` в контейнер Docker. Аналогично нужно смонтировать каталог `target`. Наконец, нужно изменить командную строку для запуска приложения и учесть в ней изменившееся имя каталога.

- ▲ Как уже отмечалось, Vagrant может использовать Docker как провайдера. Сгенерируйте соответствующее окружение, опираясь на этот пример.

2.5.5. Docker Machine

Vagrant используется для установки окружений на ноутбук разработчика. Для развертывания кроме Docker Vagrant может также использовать простые сценарии командной оболочки. Однако это решение плохо подходит для развертывания рабочих окружений. Утилита Docker Machine [34] создана специально для Docker. Она поддерживает намного больше решений виртуализации и даже нескольких поставщиков облачных услуг.

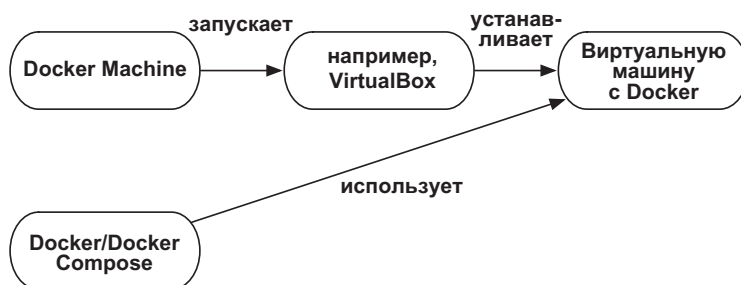


Рис. 2.6. Docker Machine

На рис. 2.6 показано, как с помощью Docker Machine можно создать окружение Docker: сначала устанавливается виртуальная машина с использованием решения виртуализации, такого как VirtualBox. Это Boot2Docker, очень легковесная версия Linux, назначение которой — служить средой времени выполнения для контейнеров Docker. Затем в эту виртуальную машину Docker Machine устанавливает текущую версию Docker. Команда `docker-machine create --driver virtualbox dev`, например, сгенерирует новое окружение с именем `dev`, выполняющееся в виртуальной машине VirtualBox.

Теперь Docker сможет взаимодействовать с этим компьютером. Для взаимодействий с сервером Docker инструмент командной строки Docker использует интерфейс REST. Поэтому достаточно определить соответствующие настройки для этого инструмента. Для этого в Linux или Mac OS X выполните команду `eval "$(docker-machine env dev)"`. В Windows PowerShell аналогичная команда имеет вид `dockermachine.exe env --shell powershell dev`, а для командной строки Windows — `docker-machine.exe env --shell cmd dev`.

То есть утилита Docker Machine значительно упрощает установку одного или нескольких окружений Docker. Все эти окружения можно администри-

ровать с помощью Docker Machine и обращаться к ним с помощью инструмента командной строки Docker. Так как Docker Machine поддерживает также такие технологии, как Amazon Cloud или VMware vSphere, эту утилиту с успехом можно применять для развертывания рабочих окружений.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

1. Описание установки Docker Machine находится по адресу: <https://docs.docker.com/machine/#installation>.
2. Docker Machine использует технологии виртуализации, такие как VirtualBox. Описание установки VirtualBox находится по адресу: <https://www.virtualbox.org/wiki/Downloads>.
3. После установки VirtualBox окружение Docker создается командой `docker-machine create --driver virtualbox dev`.
4. Команда `docker-machine env dev` объяснит, как получить доступ к окружению. Для настройки доступа достаточно выполнить команду `eval "$(docker-machine env dev)"` (в Linux/Mac OS X), `docker-machine.exe env --shell powershell dev` (в Windows PowerShell) или `docker-machine.exe env --shell cmd dev` (в командной строке Windows). Параметр `--shell` «придет на помощь» в случаях, если командная оболочка определяется неправильно.
5. Скопировать проект примера можно с помощью инструмента командной строки Git: `git clone https://github.com/ewolff/user-registration-v2.git`.
6. После этого создайте Docker-образ с Java, выполнив команду `docker build -t java java` в каталоге `docker`.
7. Аналогично можно создать образ Docker с приложением, выполнив команду `docker build -t user-registration user-registration` в каталоге `docker`.
8. После этого приложение запускается командой `docker run -p 8080:8080 -d user-registration`. Она подключит порт 8080 контейнера Docker к порту 8080 сервера.
9. С помощью команды `docker-machine ip dev` узнайте IP-адрес окружения. Приложение будет принимать запросы, поступающие в порт 8080, и его можно вызвать из браузера, обратившись по адресу URL `http://<ip>:8080/`.

10. На странице <https://docs.docker.com/machine/get-started-cloud/> вы узнаете, как использовать Docker Machine с облачными технологиями и как запустить в облачном окружении пример приложения. Изменится только команда установки окружения `docker-machine create`.
11. По окончании экспериментов выполните команду `docker-machine rm`, чтобы удалить окружение. Это особенно важно при использовании облачного окружения, чтобы снизить расходы на оплату услуг.

2.5.6. Сложные конфигурации Docker

Пример приложения в этой главе — особый случай. Это очень простое приложение, не использующее базы данных и состоящее из единственного контейнера. Более сложные окружения включают множество контейнеров, взаимодействующих друг с другом. Контейнер Docker — это некий компонент, например база данных или веб-приложение, обслуживающий клиентов вместе с другими компонентами.

Существуют различные способы обеспечить возможность взаимодействия между контейнерами Docker.

- Контейнер может обмениваться данными через сетевые порты. В примере приложения порт используется для приема веб-запросов. Такие порты могут использоваться не только системой-носителем, но и другими контейнерами. Этот подход требует установки соединений между контейнерами.
- Разумеется, контейнеры могут пользоваться каталогами в системе-носителе. Аналогично контейнеры могут использовать для взаимодействия разделы в файловой системе носителя.

То есть контейнеры Docker можно применять для реализации отдельных компонентов, которые способны обмениваться данными через сетевые порты или общие каталоги в файловой системе. В главе 8 этот подход приводится для создания сложной системы, включающей пример приложения и механизм анализа журналов приложения.

Docker Registry

Образ Docker включает данные для виртуального жесткого диска. Docker Registry позволяет сохранять и загружать образы Docker. Это дает возмож-

ность сохранить образы Docker, полученные в ходе сборки, и впоследствии скопировать их на серверы. Благодаря эффективности хранения образов легко можно создавать сложные, распределенные комплексы. Кроме того, многие поставщики облачных услуг предлагают поддержку прямого запуска контейнеров Docker.

Docker в кластере

В сценариях, описывавшихся до сих пор, Docker использовался для развертывания контейнеров на сервере. В итоге Docker превратился в инструмент автоматизации установки программного обеспечения. При таком подходе изменяется только программное обеспечение, находящееся выше уровня операционной системы: вместо отдельных процедур и приемов установки программного обеспечения с применением сценариев или инструментов, таких как Chef, теперь используется Docker. Но в основе все так же действует давно известная операционная система Linux.

Иногда требуется развернуть систему контейнеров Docker непосредственно в кластере. В этом случае можно, например, запустить несколько экземпляров контейнеров в зависимости от нагрузки и требований к доступности. И Docker способен решать такие задачи, которые большинство компаний уже решили с применением средств виртуализации.

В таких ситуациях используются следующие технологии.

- Apache Mesos [35] — планировщик. Управляет кластером серверов и распределяет задания между серверами. Mesosphere [36] позволяет запускать контейнеры Docker с помощью планировщика Mesos. Mesos также позволяет решить много других задач.
- Kubernetes [37] аналогично поддерживает выполнение контейнеров Docker в кластере. Однако использует иной подход, нежели Mesos. Kubernetes предлагает услуги распределенных контейнеров в кластере. Контейнеры Kubernetes — это связанные контейнеры Docker, выполняющиеся на физическом сервере. Для установки Kubernetes требуется лишь простая операционная система, а управление кластером реализует Kubernetes. В основе Kubernetes лежит система, используемая внутри Google для администрирования контейнеров Linux.
- CoreOS [38] — очень легковесная серверная операционная система. Через etcd поддерживает распределенные конфигурации для создания кластеров. Демон fleetd позволяет развертывать службы в кластере

и обеспечивает избыточную установку, отказоустойчивость, поддержку зависимостей и развертывание на одном определенном узле. Все службы развертываются как контейнеры Docker, тогда как сама операционная система остается практически неизменной. CoreOS можно использовать как основу для Kubernetes.

- Docker Machine [39] поддерживает возможность установки Docker на различные виртуальные и облачные системы (раздел 2.5.5). Docker Compose (описывается в следующем разделе) предоставляет возможность настраивать большое количество контейнеров Docker и соединений между контейнерами. Docker Swarm [40] позволяет объединять в кластеры серверы, сгенерированные с помощью Docker Machine. Конфигурация системы Docker Compose определяет, какие части системы должны быть распределены в кластере и как они должны быть распределены.

2.5.7. Docker Compose

Docker Compose [41] позволяет определять контейнеры Docker, совместно предоставляющие услуги. Файлы Docker Compose имеют формат YAML.

Рассмотрим конфигурацию примера приложения и решения мониторинга Graphite, которое более подробно обсуждается в разделе 8.8.

В листинге 2.10 демонстрируется конфигурация мониторинга примера приложения средствами Graphite. Она определяет следующие службы.

- **carbon** — сервер, сохраняющий данные мониторинга приложения. Элемент **build** указывает на то, что в подкаталоге **carbon** имеется файл **Dockerfile**, определяющий, как генерируется образ Docker для службы. Элемент **port** связывает порт 2003 контейнера с портом 2003 системы-носителя, где выполняется контейнер. Этот порт может использоваться приложением для сохранения информации в базе данных.
- **graphite-web** — веб-приложение, позволяющее пользователю анализировать данные. Оно доступно через порт 8082 в системе-носителе, который связан с портом 80 контейнера Docker. Элемент **volumes_from** определяет, что жесткий диск с базой данных Whisper в контейнере **carbon** также доступен в этом контейнере. Whisper — это библиотека, реализующая механизм хранилища и позволяющая сохранять и извлекать данные мониторинга.
- Наконец, контейнер **user-registration** содержит само приложение. Через порт контейнера **carbon** приложение предоставляет данные мониторинга.

га, поэтому эти два контейнера связаны. Благодаря этому контейнер `user-registration` может обращаться к контейнеру `carbon` по имени хоста `carbon`.

Листинг 2.10. Конфигурация Docker Compose для примера приложения с решением мониторинга

```
carbon:
  build: carbon
  ports:
    - "2003:2003"
graphite-web:
  build: graphite-web
  ports:
    - "8082:80"
  volumes_from:
    - carbon
user-registration:
  build: user-registration
  ports:
    - "8083:8080"
  links:
    - carbon
```

В отличие от конфигурации Vagrant здесь отсутствует контейнер Java, содержащий только установку Java. Причина в том, что Docker Compose поддерживает только контейнеры, которые действительно предлагают услуги. Поэтому теперь этот базовый образ загружается из Интернета.

В результате создается система с тремя контейнерами, взаимодействующими друг с другом (рис. 2.7) через сетевые соединения или общие файловые системы.

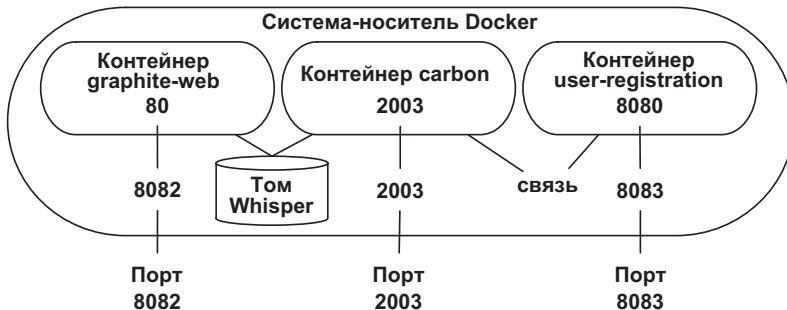


Рис. 2.7. Окружение, определяемое конфигурацией Docker Compose

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

1. Сначала сгенерируйте окружение Docker, например, с помощью Docker Machine (см. раздел 2.5.5). После этого должна быть доступна команда `docker`.
2. Установите Docker Compose — см. <https://docs.docker.com/compose/install/>.
3. Скопируйте проект примера из репозитория с помощью инструмента командной строки Git: `git clone https://github.com/ewolff/user-registration-V2.git`.
4. Перейдите в подкаталог `graphite`.
5. С помощью команды `docker-compose` и конфигурационного файла Docker Compose создайте систему. В результате будут сгенерированы соответствующие образы.
6. Запустите команду `docker-compose up`. Docker Compose использует те же настройки, что и инструмент командной строки Docker. Она также способна работать вместе с Docker Machine, поэтому с равным успехом может использоваться для запуска систем, созданных в локальной виртуальной машине или где-нибудь в облаке.

Теперь система доступна через указанные порты.

Предложения по дальнейшим экспериментам.

- ▲ Объясните, как именно работает том файловой системы, общий для контейнеров `user-registration` и `graphite-web`. Для этого загляните в файлы `Dockerfile`. Где именно определяется том?
- ▲ Исследуйте работу Docker Registry. Загрузите пример приложения в репозиторий Docker и запустите его оттуда.
- ▲ Дополните настройки, включающие Docker Machine и Docker Compose, возможностью использования Docker Swarm (см. ссылку [40]) для запуска приложения в кластере. Поскольку Docker Swarm выполняется в инфраструктуре Docker Machine, а конфигурация Docker Compose включает настройки Docker Swarm, это не должно быть очень сложно.
- ▲ Система на основе конфигурации Docker Compose может также выполняться под управлением Mesos/Mesosphere, Kubernetes или

CoreOS (см. ссылки [35], [36], [37] и [38]). Однако эта инфраструктура существенно отличается от простой инфраструктуры Docker, поэтому для ее развертывания потребуется приложить больше усилий.

- ▲ Такие решения, как Amazon Cloud, позволяют через EC2 Container Service [42] выполнять контейнеры Docker. Это также может служить основой для системы Docker.

2.6. Неизменяемый сервер

Основной задачей таких решений, как Chef, является обеспечение идемпотентности (раздел 2.3): как бы часто ни запускался сценарий установки, результат всегда должен быть неизменным. То есть сценарии описывают желаемое состояние серверов. В ходе установки выполняются шаги, необходимые для достижения этого состояния.

2.6.1. Недостатки идемпотентности

Однако этот подход имеет ряд недостатков.

- Описание конечного состояния сервера может оказаться сложнее определения необходимых шагов.
- Описания могут быть неполными. Когда сценарий не содержит информации о ресурсе, например файле или пакете, этот ресурс не может быть приведен в требуемое состояние. Более того, когда этот ресурс присутствует, но не находится в требуемом состоянии, это может привести к ошибкам.
- Зачастую серверы постоянно обновляются в ходе работы до текущей конфигурации, но никогда не создаются заново. В таком случае может оказаться невозможным создать совершенно новый сервер из-за постоянных обновлений, которые не могут быть выполнены в текущей установке — например, потому что к этому времени они были удалены из сценариев. Эта ситуация опасна тем, что конфигурация рабочего окружения становится совершенно неясной, и поэтому очень непросто установить новый сервер со всеми изменениями.

По этим причинам лучше, когда сервер всегда и полностью создается заново. В этом случае можно гарантировать соответствие сервера всем требованиям. На этой идее основываются неизменяемые серверы. Сервер всегда создается как одно целое путем установки программного обеспечения в базовый образ. Когда назревает необходимость внести изменения в конфигурацию или в процедуру установки, создается полностью новый образ. Установленный сервер никогда не корректируется и не изменяется. Это гарантирует возможность точного воспроизведения серверов в любой момент, и они всегда будут иметь одну и ту же правильную конфигурацию.

Однако на первый взгляд кажется, что воплощение идеи неизменяемых серверов требует больших трудозатрат — в конце концов, каждый раз приходится создавать сервер заново. Но в сравнении с другими трудозатратами на разных этапах конвейера непрерывного развертывания эти трудозатраты не выглядят значительными — трудозатраты на тестирование, например, куда больше.

2.6.2. Неизменяемый сервер и Docker

Оптимизация, достигаемая с применением Docker, особенно заметна при использовании неизменяемых серверов. Например, когда изменяется только файл конфигурации и копирование этого файла в образ является последним шагом в Dockerfile, создание новых образов Docker происходит очень быстро. Кроме того, при таком подходе практически не расходуется дисковое пространство, поскольку на каждом шаге установки Docker использует один и тот же образ. Соответственно, Docker может повторно использовать базовые образы, сохранившиеся от предыдущих установок. Новый образ приходится создавать только на последнем шаге, когда копируется конфигурационный файл. Однако этот образ получится не очень большим и будет сгенерирован быстро.

Все сказанное выше означает, что Docker можно объединить с идеей неизменяемых серверов и тем самым гарантировать установку программного обеспечения в конкретном состоянии и в то же время упростить процедуру установки по сравнению с подходами, главной целью которых является идемпотентность. Конечно, идею неизменяемых серверов можно также реализовать без применения Docker или с комбинацией Docker и Chef.

2.7. Инфраструктура как код

Инструменты, описанные к данному моменту, в отличие от классических подходов изменяют характер инфраструктуры. Фактически инфраструктура превращается в код — как рабочий код приложения. По этой причине широкое распространение получил термин *инфраструктура как код* (infrastructure as code). Инфраструктура генерируется не в ходе сложного процесса, выполняемого вручную, а в ходе автоматизированной процедуры. Это дает ряд преимуществ.

- Практически исключены ошибки, которые могут возникнуть в ходе создания окружений. Каждое окружение является результатом работы одного и того же программного обеспечения. Соответственно, все окружения должны выглядеть одинаково. Это предоставляет дополнительные гарантии безопасности и надежности при развертывании рабочего окружения.
- Гарантируется полное совпадение тестового и рабочего окружений, вплоть до уровня правил брандмауэра и топологии сети. Именно в этой области большинство тестовых окружений отличаются от рабочих окружений. Когда подход «инфраструктура как код» проводится в жизнь последовательно и неуклонно, таких различий можно избежать и повысить прогностическую ценность тестов. Следовательно, меньше ошибок выявится на этапе эксплуатации.
- Если во время тестирования возникают проблемы с окружением или когда окружение изменяется по неосторожности, исправить или восстановить окружение можно только вручную. При использовании подхода «инфраструктура как код» окружение можно просто удалить и заменить новым. Как вариант, можно реализовать автоматическое исправление окружения.
- Инфраструктура может иметь версии и изменяться синхронно с изменением программного обеспечения. Это гарантирует соответствие между инфраструктурой и программным обеспечением. Когда конкретное программное обеспечение потребует изменений в инфраструктуре, подобный механизм обеспечит внесение этих изменений.
- Можно четко определить процесс изменения инфраструктуры: это позволит увидеть и оценить все изменения в инфраструктуре. Данный

механизм гарантирует также документирование всех изменений и их контролируемость.

- Кроме того, *инфраструктура как код* помогает вести учет устанавливаемого программного обеспечения. Каждую программу, каждый компонент можно найти в правилах установки. Это упрощает инвентаризацию — требование на установку каждого отдельного компонента можно найти в конфигурации. Следовательно, при таком подходе можно решать проблемы, для которых обычно требуется использовать систему управления конфигурациями (Configuration Management System, CMS). Также централизованно можно реагировать на необходимость установки обновлений, например, из-за обнаруженных проблем с безопасностью.
- Обычно количество доступных окружений ограничено. Кроме того, окружениями нужно управлять: устанавливать новое программное обеспечение или внедрять вновь поступившие тестовые данные. Автоматизация позволит выполнить установку окружения без особых усилий. А когда серверы можно гибко комплектовать благодаря виртуализации, появляется возможность сгенерировать неограниченное (в принципе) количество окружений. Это особенно удобно для тестирования, так как позволяет параллельно тестировать сразу несколько версий и проблем. В идеале окружение должно выбираться через портал и предоставляться автоматически. Аналогично в часы пиковой нагрузки можно запускать дополнительные окружения. Например, в конце отчетного года установить дополнительные фермы серверов и удалить их впоследствии.
- Эксплуатация постоянно требует финансовых затрат. В то же время количество приложений, а значит, и количество систем постоянно растет — отчасти из-за того, что виртуализация обходится дешевле покупки аппаратного обеспечения. Чтобы обеспечить бесперебойную работу увеличивающегося и усложняющегося «зоопарка» силами имеющейся команды, необходимо повышать эффективность труда. Автоматизация и инфраструктура как код — это инструменты, которые делают возможным такое повышение эффективности.
- Чтобы получить настоящие выгоды от перечисленных преимуществ, изменения в инфраструктуру должны вводиться только через изменение кода, определяющего инфраструктуру. Путь к этой цели может оказаться трудным и тернистым. Поэтому имеет смысл предпринять попытку уменьшить сложность окружения.

2.7.1. Тестирование инфраструктуры как кода

Поскольку инфраструктура преобразована в код, к ней применимы обычные правила, действующие для кода: например, можно написать тесты для проверки этого кода. Эта распространенная практика в отношении бизнес-кода должна также применяться к коду автоматизации инфраструктуры. Соответственно, для решения этой задачи имеются специальные инструменты [43]. Ключевой термин — инфраструктура, управляемая тестами.

Serverspec

Serverspec [44] дает возможность на практике реализовать тесты. Эта технология позволяет разрабатывать тесты для проверки состояния, в котором оказывается сервер после установки. Тесты выполняются через SSH и поэтому не требуют установки сложного программного обеспечения на тестируемый сервер. Кроме того, состояние серверов можно оценивать независимо от особенностей реализации автоматизации инфраструктуры.

Test Kitchen

Test Kitchen [45] — это решение, специально разработанное для Chef. Оно позволяет тестировать сценарии Chef с помощью разных решений виртуализации и также поддерживает разработку кода Chef, управляемую тестами, то есть тест может быть сгенерирован до реализации автоматизации инфраструктуры.

ChefSpec

Кроме того, с помощью ChefSpec [46] можно выполнять модульное тестирование. В данном случае автоматические процедуры в действительности не выполняются, они только имитируются. Это позволяет быстро выполнять тесты и быстро получать обратную связь. Однако, поскольку действующие серверы при этом не устанавливаются, такие тесты имеют довольно ограниченную прогностическую ценность. Тем не менее полезно с самого начала, хотя бы на скорую руку, проверить наличие и доступность всех необходимых ресурсов. Это помогает убедиться в синтаксической корректности конфигурации, а также выявить и устранить возможные проблемы на раннем этапе.

2.8. Платформа как услуга

Решения, такие как Docker, Chef или Puppet, имеют одно общее свойство: они автоматизируют установку сложных приложений и создают отдельный стек технологий. Для приложений на Java, например, в операционную систему сначала устанавливается среда выполнения Java, а затем сервер приложений или веб-сервер. Впоследствии на этом сервере развертывается приложение. Также могут устанавливаться дополнительные службы, необходимые приложению, такие как базы данных.

В принципе к этой проблеме можно подойти с другой стороны: когда окружение стандартизовано — то есть всегда используется один и тот же сервер приложений и база данных, — решить ее намного проще, потому что достаточно организовать стандартизованное окружение, куда затем можно установить приложение. Именно так действует подход с названием «Платформа как услуга» (Platform as a Service, PaaS): предоставляется стандартная платформа, где можно развернуть приложение. Это значительно упрощает автоматизацию развертывания. Часто достаточно одной команды в командной строке. С другой стороны, этот подход уменьшает гибкость. Внесение изменений в платформу перестает быть простой задачей.

На рынке существуют разные предложения PaaS.

- Cloud Foundry [47] — открытый проект, который можно установить в своем вычислительном центре или использовать в общедоступном облаке. Вокруг Cloud Foundry образовалась целая «экосистема» — появилось большое количество расширений и операторов, предлагающих общедоступные услуги на основе Cloud Foundry. Для поддержки разных языков программирования Cloud Foundry использует так называемые комплектующие (Buildpacks). В настоящее время существуют комплектующие для Java, Node.js, Ruby и Go. Для других языков можно использовать собственные комплектующие или попробовать использовать Heroku. В *платформы как услуги*, доступные для приложений, могут также интегрироваться базы данных.
- Heroku [48] доступна как PaaS только в общедоступном облаке. Посредством уже упомянутых комплектующих (Buildpacks) она поддерживает большое количество языков: помимо Ruby, Node.js, PHP, Python, Go, Scala, Clojure и Java имеются также комплектующие для таких редких языков, как EMACSLisp, развиваемый сообществом. Поддержка расширений позволяет включать в услуги базы данных и другие инстру-

менты, например средства для анализа файлов журналов. Поскольку Heroku действует в Amazon Cloud, на серверы в Amazon Cloud также можно устанавливать другое программное обеспечение и использовать его в программах Heroku.

- Google App Engine [49] — общедоступная облачная платформа как услуга, поддерживающая PHP, Python, Java и Go. Дополнительно включает поддержку базы данных SQL на основе MySQL и простую базу данных NoSQL.
- Amazon Elastic Beanstalk [50] использует HTTP-сервер Apache для поддержки Node.js, PHP, Go, Python, Ruby, .NET и Apache Tomcat для Java; также включает поддержку Docker. Служба работает в Amazon Cloud и поэтому позволяет использовать разнообразные базы данных и другие программные компоненты, а также устанавливать собственные серверы и запускать их в Amazon Cloud. Это дает возможность сконструировать практически любое решение.
- Microsoft Azure App Service [51] поддерживает .NET, Node.js, Java, PHP, Python и Ruby. Диапазон возможностей дополняет поддержка разных баз данных, таких как Oracle или Microsoft SQL Server. Поскольку услуга действует в Azure, имеется возможность запустить любой сервер, а кроме того, эта технология позволяет запустить практически любое приложение.
- OpenShift [52] — предложение компании Red Hat. Помимо услуги в общедоступном облаке предлагается также продукт, который вы устанавливаете в собственном вычислительном центре. OpenShift поддерживает Haskell, Java, PHP, Node.js, Ruby, Python и Perl. Для приложений на Java поддерживаются Tomcat, JBoss и Vert.x. Также имеется поддержка баз данных, таких как MongoDB, PostgreSQL и MySQL.

Эти технологии дают большие преимущества, особенно когда приложение должно быть доступно в веб: они обеспечивают простоту масштабирования приложений, а многие поставщики услуг имеют также вычислительные центры, разбросанные по всему миру, благодаря чему достигается высочайший уровень доступности. Это означает, что нет необходимости создавать собственный вычислительный центр и прокладывать свои линии связи с Интернетом. Часто это помогает снизить затраты на эксплуатацию. Развертывание приложений во внешних вычислительных центрах является более удобным решением по разным причинам — например, из-за более высокого уровня безопасности данных. Большинство

решений не дают возможности установки в собственном вычислительном центре – из списка выше только OpenShift и Cloud Foundry являются исключениями. Кроме того, установка таких окружений сложна и трудоемка. Поэтому в простых случаях предпочтительнее использовать Puppet, Chef или Docker. Кроме того, существуют решения, такие как Flynn [53] и Dokku [54], которые реализуют простые облачные решения PaaS на основе Docker.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Решения, перечисленные выше, предлагают бесплатный доступ для опробования и руководства, которые помогут сделать первые шаги. Пример приложения – это простое приложение на Java, однако оно требует поддержки Servlet 3.0. Зарегистрируйтесь на сайте поставщика облачных услуг по своему выбору и попробуйте запустить приложение в его окружении.

2.9. Хранение информации и базы данных

Базы данных представляют определенную проблему в ходе комплектования инфраструктуры, поэтому данный раздел целиком будет посвящен этой теме.

Реляционные базы данных имеют схему. Она определяет, как должны структурироваться данные, которые предполагается хранить в базе. Поэтому после установки базе данных нужно передать схему. Конечно, это можно сделать с помощью сценария, однако такой подход имеет существенный недостаток: начальной точкой для него является пустая база данных. Как следствие, эта стратегия не может применяться, когда в базе уже имеются данные, накопленные в процессе эксплуатации. Эта проблема подобна проблеме со сценариями установки, описанной в разделе 2.2: не имеет смысла слепо устанавливать программное обеспечение, следует выбирать подходы, позволяющие иметь дело с разными начальными состояниями, в данном случае базы данных, и приводить их все к желаемому конечному состоянию. Если имеется старая версия базы данных, она должна быть обновлена соответственно требованиям. Такое обновление может потребовать миграции данных – например, заполнить новый столбец значениями по умолчанию. Однако, если версия базы данных уже актуальна, такая миграция не должна выполняться.

2.9.1. Обработка схем

Чтобы получить возможность изменить схему базы данных, требуется изменить все приложения, обращающиеся к данным в ней. Поэтому, если имеется несколько приложений, обращающихся к одной и той же базе данных, и некоторые из них не развиваются дальше, изменение схемы становится практически невозможным. То есть общего доступа к базам данных следует избегать, особенно в сценариях, когда программное обеспечение активно изменяется и используется методология непрерывного развертывания.

Для подобных случаев существуют инструменты, действующие следующим образом.

- Внутри базы данных сохраняется номер версии. Номер версии однозначно определяет версию схемы базы данных.
- Для перевода базы данных из текущей версии в следующую создаются соответствующие сценарии. Эти сценарии вносят необходимые изменения в схему, а также корректируют сами данные.
- При необходимости создаются сценарии, восстанавливающие старую версию базы данных из текущей с целью отмены изменений. Подобные сценарии полезны, например, когда тестовая база данных имеет слишком новую версию, чтобы ее можно было использовать для проверки исправлений, внесенных в более старую версию программного обеспечения.

Такой инструмент позволяет определить, какие операции необходимо выполнить для миграции. Когда обнаруживается, что схема базы данных имеет версию 38, а требуется версия 42, сценарий должен выполнить миграцию версии 38 до версии 39, затем до версии 40, 41 и, наконец, 42. Если база данных абсолютно пуста, все сценарии могут выполняться друг за другом.

Вот некоторые из инструментов, поддерживающих такой подход.

- Active Record Migrations [55] — часть Ruby on Rails, предлагает поддержку описанного подхода. Изменения описываются на Ruby DSL. Этот инструмент заслуживает особого упоминания, потому что был одним из первых, основанных на этом подходе.
- Flyway [56] — реализует аналогичный подход на Java. Сценарии могут быть реализованы на SQL и на Java.
- Liquibase [57] — также реализован на Java, но вместо SQL использует свой предметно-ориентированный язык.

Кроме перечисленных существует еще много похожих инструментов, способных интегрироваться с популярными окружениями разработки программного обеспечения.

Однако проблема со схемами свойственна только реляционным базам данных. Альтернативная возможность — использовать базы данных NoSQL. Они обладают большей гибкостью в отношении схем, фактически позволяя хранить произвольные данные. То есть в них отсутствует жесткое определение схемы, поэтому его не приходится изменять. Не следует недооценивать это преимущество: действительно существуют проекты, использующие базы данных NoSQL вместо реляционных, именно по этой причине. Миграция данных и изменение схемы являются настоящей проблемой в реляционных базах данных, особенно когда в них хранятся большие объемы данных.

2.9.2. Тестовые и базовые данные

Кроме схем существуют также другие проблемы, связанные с базами данных: пустая база данных не имеет большой ценности. Обычно базы данных содержат некоторые базовые данные, для тестов — тестовые данные, а для рабочего окружения — рабочие данные.

Базовые данные можно сгенерировать с помощью сценария — эти сценарии также должны уметь обрабатывать ситуацию, когда база данных уже содержит данные. Соответственно, базовые данные генерируются, например, посредством сценариев соответствующих инструментов, реализующих миграцию схемы базы данных. То же верно в отношении тестовых данных.

Как вариант, роль тестовых данных могли бы выполнять резервные копии из рабочего окружения. Конечно, при этом не должны нарушаться требования к защищенности данных. Кроме того, тесты должны приспосабливаться к данным и действовать последовательно на всех этапах. Когда данные изменяются в рабочем окружении, это может привести к отказам на этапе тестирования.

Поэтому, в зависимости от ситуации, следует определить конкретный набор тестовых данных, адаптированный для нужд тестирования. В этом случае гарантируется соответствие тестовых данных сценариям тестирования. Однако, когда задачей является выполнение миграции старых данных, тест следует запускать как можно раньше и с как можно большим объемом старых данных, потому что такие наборы данных скрывают много сюрпризов, которые не обнаруживаются в тестовых данных, созданных искусственно.

Например, нет ничего необычно в том, что старые данные содержат некоторые наборы данных дважды или наборы данных, не соответствующие схеме, — по сути эти наборы являются недействительными. Такие наборы данных особенно ценны для тестирования, потому что могут вызывать проблемы в программном обеспечении. Соответственно, в таком тестовом сценарии должна иметься цель для запуска тестов с реальными данными из рабочего окружения на раннем этапе.

Наконец, базы данных могут проявлять разное поведение в зависимости от объема хранящихся данных. То есть миграция схемы может протекать без проблем с сокращенным набором данных, но терпеть неудачу в рабочей системе просто из-за большого объема данных. Если такая проблема обнаружится незадолго до запланированного выпуска новой версии, будет слишком поздно. Поэтому миграция данных должна тестироваться вовремя и с реальными объемами данных.

Кроме этих, сугубо технических подходов к работе с базами данных имеется также ряд аспектов более концептуального свойства. Так, в разделе 11.5 обсуждаются подходы к организации баз данных с архитектурной точки зрения и еще раз объясняется важность баз данных для методологии непрерывного развертывания.

2.10. В заключение

В этой главе были представлены разные технологии автоматизации инфраструктуры.

- *Chef* позволяет создавать новые окружения. Для этого достаточно просто описать желаемое состояние окружения. С помощью описания *Chef* приводит окружение в это состояние. Такой подход предотвращает появление многих проблем, характерных для обычных сценариев установки. Кроме того, существует много готовых сценариев.
- *Knife* и *Chef Server* можно использовать для запуска и установки сервера простым вызовом инструмента командной строки.
- *Vagrant*, напротив, отлично подходит для сборки окружений на компьютерах разработчиков.
- *Docker* представляет интересное альтернативное решение: кроме высокой эффективности по сравнению с решениями виртуализации контей-

неры Docker позволяют также устанавливать программное обеспечение с помощью простых сценариев. Даже установка контейнеров Docker в кластере выполняется относительно просто.

- *Docker Machine* позволяет устанавливать и выполнять контейнеры Docker практически на любом сервере.
- *Docker Compose* дает возможность установить и запустить несколько взаимосвязанных контейнеров Docker.
- Все эти инструменты превращают инфраструктуру в код, а это имеет далеко идущие последствия.
- Реляционные базы данных более сложны в обслуживании и сопровождении.

С помощью непрерывного развертывания окружение для каждого этапа в конвейере развертывания создается, по сути, нажатием одной кнопки. Фактически только на этой основе можно реализовать конвейер.

Ссылки

1. <https://www.virtualbox.org/>
2. <https://www.vagrantup.com/>
3. <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> (имеется перевод на русский язык первой версии этой книги: <https://git-scm.com/book/ru/v1> — Примеч. пер.).
4. <https://puppet.com/>
5. <https://www.ansible.com/>
6. <https://www.ansible.com/get-started>
7. <https://github.com/ansible/ansible-examples/tree/master/tomcat-standalone>
8. <https://saltstack.com/>
9. <https://docs.saltstack.com/en/latest/topics/tutorials/walkthrough.html>
10. <https://docs.chef.io/berkshelf.html>
11. <https://github.com/>
12. <https://supermarket.chef.io/cookbooks>

13. <https://www.virtualbox.org/>
14. https://docs.chef.io/install_server.html
15. <https://learn.chef.io/>
16. <https://www.vagrantup.com/>
17. <https://www.virtualbox.org/>
18. <https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins>
19. <http://www.vagrantbox.es/>
20. <https://github.com/jedi4ever/veewe>
21. <https://www.packer.io/>
22. <https://www.virtualbox.org/>
23. <https://www.vagrantup.com/>
24. <https://github.com/fgrehm/vagrant-cachier>
25. <https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins>
26. <https://www.packer.io/>
27. <https://github.com/devopsgroup-io/vagrant-hostmanager>
28. <https://martinfowler.com/bliki/PhoenixServer.html>
29. <https://www.docker.com/>
30. <https://github.com/ewolff/user-registration-V2>
31. <https://git-scm.com/>
32. <https://github.com/ewolff/user-registration-V2>
33. <https://git-scm.com/>
34. <https://docs.docker.com/machine/>
35. <http://mesos.apache.org/>
36. <https://mesosphere.com/>
37. <https://kubernetes.io/>
38. <https://coreos.com/>
39. <https://docs.docker.com/machine/>
40. <https://docs.docker.com/swarm/>
41. <https://docs.docker.com/compose/>

42. <https://aws.amazon.com/documentation/ecs/>
43. Stephen Nelson-Smith: *Test-Driven Infrastructure with Chef: Bring Behavior-Driven Development to Infrastructure as Code*, O'Reilly, 2nd Edition 2013, ISBN 978-1-44937-220-0.
44. <http://serverspec.org/>
45. <http://kitchen.ci/>
46. <http://docs.chef.io/chefspec.html>
47. <https://www.cloudfoundry.org/>
48. <https://www.heroku.com/>
49. <https://cloud.google.com/appengine/docs>
50. <https://aws.amazon.com/elasticbeanstalk/>
51. <https://azure.microsoft.com/ru-ru/?b=16.47>
52. <https://www.openshift.com/>
53. <https://flynn.io/>
54. <https://github.com/dokku/dokku>
55. http://edgeguides.rubyonrails.org/active_record_migrations.html
56. <https://flywaydb.org/>
57. <http://www.liquibase.org/>

Часть II

КОНВЕЙЕР НЕПРЕРЫВНОГО РАЗВЕРТЫВАНИЯ

Следующие главы подробно описывают каждый этап конвейера непрерывного развертывания.

- ✓ **Глава 3** описывает этап выпуска новой версии, включая модульные тесты, автоматизированный анализ и проверку качества программного кода, инструменты сборки и репозитории.
- ✓ **Глава 4** подробнее описывает приемы автоматизации приемочных испытаний.
- ✓ **Глава 5** посвящена автоматизации тестирования пропускной способности.
- ✓ **Глава 6** обсуждает особенности исследовательских испытаний вручную.
- ✓ **Глава 7** описывает приемы и инструменты для развертывания.
- ✓ Аспекты, наиболее важные для эксплуатации, обсуждаются в **главе 8**.

3

Автоматизация сборки и непрерывная интеграция

Эта глава написана Бастианом Спаннебергом (Bastian Spanneberg). Он ведущий инженер в Instana и занимается разработкой архитектуры платформ и автоматизацией. Его можно найти в Twitter: *@spanneberg*.

3.1. Введение

На техническом уровне применение методологии непрерывного развертывания заключается исключительно в автоматизации. Первое требование, которое нужно выполнить на пути создания конвейера непрерывного развертывания, — автоматизация процесса сборки и инфраструктуры. Эти темы рассматриваются в данной главе.

Автоматизированная сборка — основа этапа выпуска и, соответственно, первого шага в конвейере непрерывного развертывания. Поэтому в разделе 3.2 подробно обсуждается тема инструментов сборки: их особенности и приемы использования. Затем, в дополнительном разделе (раздел 3.3), обсуждаются модульные тесты. Модульные тесты являются важным инструментом для сбора надежной информации о функционировании и стабильности программного обеспечения в процессе сборки. Раздел 3.4 посвящен непрерывной интеграции, то есть непрерывной сборке, интеграции и тестированию программного обеспечения. Для организации непрерывной интеграции широко используется сервер Jenkins, который будет представлен в этом разделе как пример конкретной технологии. Статический анализ качества кода обсуждается в разделе 3.5. Здесь детально рассматривается широко используемый для этих целей инструмент SonarQube. Раздел демонстрирует, как этот инструмент интегрируется в процесс сборки и в окружение непрерывной интеграции. На примере Artifactory последний раздел этой главы (раздел 3.6) исследует тему использования репозиторий для хранения артефактов и их роль в контексте непрерывной интеграции и непрерывного развертывания.

3.1.1. Автоматизация сборки: пример

В разделе П.2 рассматривался пример компании «Big Money Online Commerce Inc.», где не использовалась методология непрерывного развертывания. Однако в этой компании настроен процесс автоматизированной сборки и сервер непрерывной интеграции. Кроме того, в ходе подготовки к переходу на методологию непрерывного развертывания был введен статический анализ качества кода для выявления проблем со сложностью кода и оценки охвата тестами. Этот анализ выдал несколько рекомендаций по улучшению качества проекта и особенно по повышению удобства его сопровождения. Результаты сборки теперь сохраняются в репозитории, что делает их доступными на любом этапе конвейера. Это создает прочный технологический фундамент для конвейера непрерывного развертывания и гарантирует необходимое качество для последующих этапов.

3.2. Автоматизация сборки и инструменты сборки

Инструменты сборки автоматизируют компиляцию и компоновку программного обеспечения. Обычно сборка программного обеспечения производится в несколько этапов, зависящих друг от друга. Полная последовательность событий зависит, например, от используемого языка программирования и целевой платформы. Тем не менее многие этапы являются общими для любого программного обеспечения, например следующие.

- Компиляция исходного кода в двоичный.
- Запуск модульных тестов и оценка их результатов.
- Обработка имеющихся исходных файлов (например, файлов конфигурации).
- Создание артефактов для последующего использования (например, файлов WAR или EAR, пакетов Debian).

Следующие этапы также часто выполняются в процессе сборки и аналогично могут быть автоматизированы с применением инструментов сборки.

- Подготовка зависимостей, например библиотек, используемых в проекте.
- Выполнение дополнительных испытаний, например приемочных и нагрузочных.

- Анализ качества исходного кода и проверка его соответствия принятым соглашениям (статический анализ кода).
- Передача сгенерированных артефактов и пакетов в центральный репозиторий.

Многие из этих задач решает среда разработки, поэтому разработчики обычно не уделяют внимания их автоматизации с помощью инструментов. Тем не менее очень важно внедрить автоматизацию всех этапов с применением специализированных инструментов, потому что это гарантирует воспроизводимость результатов, а также изоляцию и независимость процесса сборки от используемых средств. В конечном счете разработчики могут пользоваться разными инструментами, но процесс сборки все равно останется унифицированным и воспроизводимым. Такая процедура сборки составляет основу для применения непрерывной интеграции и, соответственно, непрерывного развертывания.

3.2.1. Инструменты сборки в мире Java

В мире Java в области автоматизации сборки в настоящее время доминируют три инструмента: Ant, Maven и Gradle. Эти инструменты используют совершенно разные подходы.

- Ant [1] следует императивному подходу. Разработчики должны позаботиться о настройке всех аспектов. Например, они должны создать каталоги для скомпилированного кода и явно обработать зависимости между отдельными этапами сборки. По сути, разработчики должны сами реализовать всю процедуру сборки, шаг за шагом.
- Maven [2] следует в противоположном направлении и использует декларативный подход. Многие аспекты процесса сборки предопределяются соглашениями, которым должны следовать проекты, использующие Maven. Примерами таких соглашений могут служить местоположение каталогов с исходным кодом и тестами, каталогов для сохранения результатов сборки и последовательность фаз сборки. Это очень упрощает настройку сборки. Однако везде, где процедура сборки отклоняется от соглашений или где требуется изменить настройки, разработчик должен явно объявить об этом. Поэтому на практике процедура сборки с применением Maven часто оказывается еще более сложной, чем с использованием других инструментов. В любом случае последовательность этапов сборки должна соответствовать жизненному циклу модели Maven, определяющей разные фазы сборки.

- Gradle [3] реализует новейший подход к применению средств сборки для Java. Этот инструмент пытается объединить лучшие черты императивного и декларативного подходов. По аналогии с Maven, многие аспекты сборки в Gradle определяются соглашениями. Однако при отклонении от соглашений Gradle, так же как Ant, позволяет разработчику полностью отказаться от них и реализовать независимые процедуры на предметно-ориентированном языке Gradle DSL или на языке программирования Groovy, лежащем в основе Gradle.

Следующие разделы более подробно описывают все три инструмента.

3.2.2. Ant

Ant — самый старый из упомянутых инструментов. Он имеет много общего с инструментом сборки Make, хорошо известным в Unix, и использует императивный подход. Это означает, что разработчик сам реализует все фазы сборки в сценариях Ant, в файлах XML и определяет зависимости между фазами. Результаты, которые должны быть достигнуты в ходе сборки, в терминологии Ant называются целями (targets) и состоят из последовательностей задач (tasks). Ant включает большое количество predefined задач — например, для обработки файлов и каталогов, компиляции исходного кода на Java или создания архивов разных типов. Кроме того, разработчик может реализовать свои задачи на Java, если имеющихся в арсенале Ant оказывается недостаточно. В проекте Ant Contrib [4] можно найти большое количество дополнительных задач, которые легко можно интегрировать в свои проекты. Ant не управляет зависимостями. Однако существует отдельное решение — Ivy [5], которое очень легко интегрируется с Ant.

Во многих компаниях до сих пор можно найти реализации очень сложных процедур сборки на Ant. В новых проектах Ant, как правило, не рассматривается в числе возможных вариантов, потому что полная реализация сборки в XML обычно довольно сложна. Поэтому в данной главе основное внимание уделяется Maven и Gradle. Но если мы говорим о непрерывном развертывании, выбор того или иного инструмента сборки не играет существенной роли. Как бы то ни было, Ant позволяет без проблем реализовать все необходимые шаги, хотя реализация получится более громоздкой, чем при использовании более современных инструментов.

Самое важное — инструмент должен позволять четко и правильно автоматизировать все необходимые шаги и непрерывно сопровождать и изменять

логику сборки, по аналогии с программным кодом, чтобы гарантировать высокое качество сценариев сборки в течение длительного времени.

3.2.3. Maven

В настоящее время Maven является, пожалуй, самым широко используемым инструментом сборки для Java — особенно в корпоративном секторе. В отличие от более раннего инструмента Ant, фактически превратившегося в стандарт, Maven не требует от разработчика описывать все необходимые задачи для компиляции, тестирования и упаковки, а также определять зависимости между фазами сборки. Вместо этого в Maven используется модель «преимущества соглашений перед конфигурацией» («convention over configuration»), поддерживающая предопределенный и универсальный жизненный цикл. Конечно, этот подход определенно ограничивает круг вариантов сборки, которые Maven может поддерживать. По умолчанию жизненный цикл Maven включает несколько предопределенных и фиксированных этапов. Внутри них выполняются все действия, необходимые для сборки классического программного обеспечения. На рис. 3.1 изображена последовательность событий, реализованная в Maven: сначала производится проверка файлов с исходным кодом и инициализируется процедура сборки. Затем осуществляются компиляция и обработка исходного кода и ресурсов. То же происходит с исходным кодом и ресурсами тестов. Далее результат упаковывается и, если необходимо, выполняется интеграционное тестирование. В конце есть возможность проверить, установить и развернуть результат сборки в репозитории артефактов. В каждой фазе могут выполняться не все предопределенные действия. Однако таким способом Maven обеспечивает скелет, на который вы можете ориентироваться, определяя свою логику сборки.

Благодаря стандартизации во всех проектах Maven используются одни и те же фазы, команды компиляции и тестирования программного обеспечения: команда `mvn package` выполняет все фазы, вплоть до упаковки и создания сборки. В отличие от нее команда `mvn test` останавливается на этапе «тестирование».

Конкретные задачи на разных этапах реализуют плагины для Maven. Выполнение разных плагинов отображается на соответствующие этапы. Для каждого этапа существуют строго определенные плагины. Посредством этапов процесс сборки в Maven можно расширить за пределы стандарта. Например, плагин, загружающий и запускающий тесты на этапе «тестирование», можно настроить на выполнение дополнительных тестов. С другой

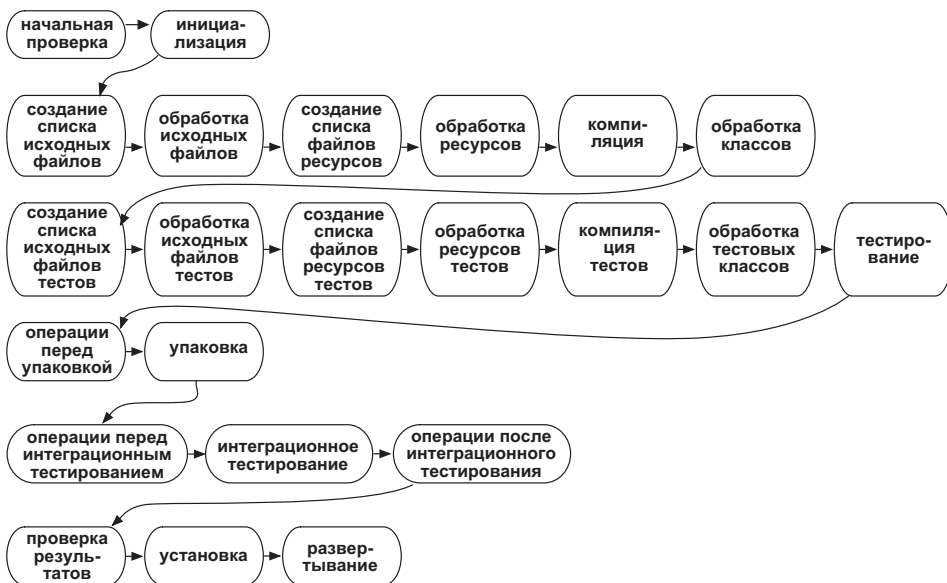


Рис. 3.1. Фазы стандартной процедуры сборки в Maven

стороны, определенные действия, выполняемые плагинами, которые в терминологии Maven называют целями (goals), можно выполнять по отдельности из командной строки.

Maven использует подход следования соглашениям. Это имеет свои преимущества — файлы с параметрами сборки получаются более компактными и управляемыми, чем аналогичные файлы Ant, потому что в Maven нет необходимости явно определять все детали. Однако в действительности файлы Maven с параметрами сборки имеют существенный размер. Одна из причин большого размера — многие настройки по умолчанию в Maven весьма консервативны. Например, плагин компилятора все еще по умолчанию использует Java 1.5, хотя текущей является версия Java 1.8, а версия 1.5 уже даже не поддерживается. Такие настройки, как правило, приходится переопределять. Кроме того, сам формат XML способствует быстрому росту размеров файлов. Однако структура файлов с параметрами сборки не меняется. Это помогает разработчикам быстро перейти на использование Maven. Им достаточно просто познакомиться с предопределенной структурой.

В листинге 3.1 показан минимальный пример объектной модели проекта Maven (Project Object Model, POM) для проекта Java 8. Модель POM

определяет, как Maven выполняет сборку. Если настройки совместно используются несколькими проектами Maven, лучшее решение состоит в том, чтобы определить так называемую родительскую модель POM. Родительская модель позволяет разработчику определить параметры для сборки нескольких проектов и встраивать индивидуальные настройки для каждого отдельного проекта. Тогда в конкретных проектах понадобится определить только настройки, уникальные для их моделей POM. Пример приложения регистрации пользователя, в частности, использует родительскую модель POM из фреймворка Spring Boot Framework.

Листинг 3.1. Базовая модель POM для Maven (pom.xml)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" ... >
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-project</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>org.junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Здесь определяется имя проекта, состоящее из идентификаторов группы и артефакта, а также номера версии. Используя эту информацию, другие компоненты могут ссылаться на данное программное обеспечение. Зависимости от других компонентов определяются аналогично. Объявление способа упаковки определяет тип артефакта, создаваемого в процессе сборки.

Кроме JAR (Java Archive — архив Java) можно также указать, например, WAR (Web Archive — веб-архив) и EAR (Enterprise Archive — корпоративный архив). Конкретная реализация упаковки (в данном случае) скрыта в плагине по умолчанию `maven-jar-plugin` [6].

Как можно заметить в примере, обработка зависимостей интегрирована непосредственно в Maven. В данном случае определена зависимость от фреймворка тестирования JUnit. В ходе сборки Maven автоматически загружает необходимые зависимости из центрального репозитория в Интернете или из внутреннего репозитория компании и добавляет в путь поиска классов. Maven также обрабатывает промежуточные зависимости, то есть зависимости зависимостей. Более подробную информацию по этой теме можно найти в документации с описанием Maven [7].

Версии и мгновенные снимки

Еще одна интересная идея Maven — использование так называемых мгновенных снимков (`snapshot versions`). Их назначение — выпуск промежуточных версий программного обеспечения между официальными выпусками для других членов команды или для компонентов, определяющих зависимость от данного программного обеспечения. Такие промежуточные версии сохраняются в специальных репозиториях артефактов, откуда могут быть получены всеми заинтересованными сторонами. Каждый раз процедура сборки мгновенного снимка — например, `0.0.1-SNAPSHOT` — затирает свои предыдущие результаты. Благодаря этому под одним и тем же номером версии всегда распространяется самый свежий мгновенный снимок. То есть программное обеспечение изменяется, а номер версии остается прежним. Как результат, инструменты сборки теряют возможность отличить одну версию от другой. Это не соответствует цели непрерывного развертывания: каждое изменение и, соответственно, каждая сборка является потенциальным и однозначно идентифицируемым кандидатом на выпуск. Каждая сборка проходит через конвейер непрерывного развертывания и в конце может даже попасть в рабочее окружение, если будет иметь достаточно высокое качество. Как следствие, использовать мгновенные снимки имеет смысл только для сборок, выполняемых за пределами конвейера непрерывного развертывания, например для создания локальных сборок, используемых отдельным разработчиком.

Однако в мире Maven никогда не стояло цели выводить мгновенные снимки в рабочие окружения. Они необходимы лишь как интеграционные арте-

факты для других коллективов и компонентов. В этом отношении Maven противоречит основной идее непрерывного развертывания.

Выпуск версий с помощью Maven

Классический способ сгенерировать в Maven версию для выпуска — задействовать плагин `release` [8]. Он выполняет ряд действий, которые разработчик мог бы выполнять самостоятельно, в ходе ручной сборки новой версии.

- В модели POM из версии удаляется окончание `-SNAPSHOT`, после чего осуществляется сборка и тестирование программного обеспечения. В результате создается версия программного обеспечения с соответствующим номером версии.
- После успешной сборки измененная модель POM копируется в систему управления версиями и отмечается тегом. Тегом обычно служит номер версии. Затем номер версии увеличивается и дополняется окончанием `-SNAPSHOT`. Это изменение также записывается в систему управления версиями.
- Затем извлекается версия, отмеченная тегом, собирается и тестируется.
- Сгенерированные артефакты (например, JAR, WAR или EAR) сохраняются в репозиторий артефактов.

Несмотря на отсутствие изменений в программном обеспечении, кроме номера версии, программное обеспечение полностью пересобирается и тестируется несколько раз. Кроме того, плагин создает в системе управления версиями две новые версии кода.

Этот подход невозможен, если каждое изменение рассматривается как потенциальная причина для выпуска новой версии, согласно идее непрерывного развертывания. Во-первых, потому что создаются две новые версии программного обеспечения, не отличающиеся ничем, кроме номера версии в модели POM. Во-вторых, программное обеспечение собирается и тестируется чаще, чем требуется, что ведет к напрасному расходованию ресурсов. Было бы желательно иметь какую-то процедуру, позволяющую однозначно идентифицировать каждую сборку без участия плагина `release`.

Один из возможных вариантов — задавать версию за границами этапа сборки. Это можно реализовать с помощью плагина `version` [9] в Maven. Он устанавливает версию проекта, заданную в параметре:

```
> mvn versions:set -DnewVersion=0.0.2  
> mvn clean deploy
```

Сервер сборки использует эту возможность, например, для получения последовательных номеров сборки и добавления их в номер версии. Отметку версий тегом в системе управления версиями можно выполнить с помощью плагина SCM [10] в Maven:

```
> mvn scm:tag
```

Если эти два элемента включить в процедуру сборки, появится возможность однозначной идентификации версии каждого изменения. Даже если модель POM определяет версию мгновенного снимка, в этом сценарии она будет использоваться разработчиками только для интеграции. Однако при этом необходимо решить, имеет ли смысл в контексте непрерывного развертывания генерировать тег с помощью SCM, потому что каждая сборка является потенциальным кандидатом на выпуск и поэтому могла бы отмечаться своим тегом. Как вариант, можно сделать частью номера версии хэш-код фиксации или сделать его доступным внутри приложения другими средствами, чтобы иметь возможность идентифицировать версию исходного кода во время выполнения.

Интересное и познавательное обсуждение мгновенных снимков, выпусков и непрерывного развертывания можно найти в статьях Акселя Фонтейна (Axel Fontaine) «Maven Releases on Steroids» [11] и «Maven Release Plug-In: The Final Nail in the Coffin» [12].

3.2.4. Gradle

Gradle [13] — это альтернатива Maven, набирающая популярность в последнее время. Создатели инструмента обозначили его цель как объединение гибкости Ant с подходом «преимущества соглашений перед конфигурацией», принятым в Maven, для объединения лучших черт обоих миров.

Gradle реализован как предметно-ориентированный язык (DSL), основанный на языке программирования Groovy [14]. Следовательно, все сценарии Gradle одновременно являются сценариями на языке Groovy, что дает разработчикам возможность встраивать код на Groovy непосредственно в сценарии сборки и таким способом реализовать практически любые возможности. Язык Groovy имеет много общего с Java. Любая программа на

Java является также допустимой программой на Groovy. Однако Groovy поддерживает более простой синтаксис.

Основными понятиями Gradle являются задачи и зависимости между ними. На их основе Gradle вычисляет направленный ациклический граф, чтобы определить порядок выполнения задач. Это необходимо, потому что этот граф может изменяться путем добавления нестандартных задач, дополнительных плагинов или изменения существующих зависимостей.

Еще одно преимущество Gradle — поддержка инкрементальной сборки: Gradle выполняет задачи сборки, только если изменился исходный код или если задача никогда не выполнялась прежде. Иначе Gradle пропускает задачу и сразу переходит к следующему шагу в графе выполнения. В большой и сложной процедуре сборки это может сэкономить немало времени. Как упоминалось выше, Gradle также поддерживает возможность расширения своих возможностей посредством плагинов. Благодаря этому разработчики могут встраивать в Gradle поддержку других языков программирования, таких как Groovy, C++ или Objective-C.

Gradle имеет очень компактный синтаксис в сравнении с двумя другими «конкурентами», Ant и Maven, благодаря отсутствию XML. В листинге 3.2 представлен пример сценария сборки из раздела с описанием Maven на языке Gradle.

Листинг 3.2. Минимальный сценарий сборки на Gradle

```
apply plugin: 'java'

archivesBaseName = 'my-project'
version = '1.0.0-SNAPSHOT'

sourceCompatibility = '1.8'
targetCompatibility = '1.8'

repositories {
    mavenCentral()
}

dependencies {
    testCompile "org.junit:junit:4.11"
}
```

В отношении настроек по умолчанию — например, используемой версии Java — Gradle также действует более интеллектуально, чем Maven. Он использует версию Java, под управлением которой был запущен. Если сценарий сборки, представленный выше, запустить под управлением виртуаль-

ной машины Java 8, информацию о версии языка можно опустить. Однако в этом случае сборка будет зависеть от установленной версии Java. Поэтому, чтобы гарантировать воспроизводимость результатов сборки, версию необходимо указывать. Также можно опустить свойство `archivesBaseName`, определяющее имя генерируемого архива. По умолчанию Gradle использует с этой целью имя каталога, в котором находится сценарий сборки.

Поскольку команды сборки могут различаться в зависимости от используемого плагина и задач, определяемых в сценарии сборки, Gradle поддерживает собственную задачу, отображающую все другие задачи, доступные в текущем проекте:

```
> gradle tasks
...
Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all
projects that depend on it.
buildNeeded - Assembles and tests this project and all projects
it depends on.
classes - Assembles classes 'main'.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles classes 'test'.
...
```

Команда возвращает список объявленных задач. Задачи могут определяться плагинами, непосредственно в сценарии сборки или в других подключаемых сценариях. Это облегчает разработчикам знакомство с порядком сборки проекта.

Gradle Wrapper

Для автоматизации сборки может пригодиться еще одна особенность Gradle: Gradle Wrapper. Она позволяет разработчику настроить версию Gradle, которая будет использоваться для выполнения файла сборки, а также включить в проект двоичные файлы Wrapper и сохранить их в системе управления версиями. Благодаря этому при извлечении проекта — на сервер сборки или на локальный компьютер другого разработчика — будет скопировано все, что необходимо для сборки: недостающие компоненты будут загружены с веб-сайта Gradle или из внутреннего репозитория. Благодаря этому не требуется устанавливать и сопровождать инструмент. Тот, кто настраивает Wrapper

в первый раз, должен установить Gradle Wrapper на локальный компьютер — после этого все остальные смогут просто пользоваться им. Так гарантируется использование всеми одних и тех же инструментов сборки и, соответственно, воспроизводимость результатов сборки.

Чтобы включить Wrapper в проект, достаточно в каталоге проекта выполнить задачу Wrapper:

```
> gradle wrapper
```

После этого в каталоге появятся два сценария, `gradlew` и `gradlew.bat`, для запуска Wrapper в Windows, а также в Linux или Mac OS X. Кроме того, будет создан новый каталог с именем `gradle`, куда будут помещены двоичные файлы Wrapper. Эти файлы можно отправить в систему управления версиями вместе с проектом. В таком случае Gradle будет доступен всем разработчикам. Если потребуется явно управлять версией в проекте, задачу Wrapper можно настроить непосредственно в файле сборки:

```
task wrapper(type: Wrapper) {  
    gradleVersion = '2.10'  
}
```

Это также существенно упрощает обновление версии инструмента, потому что версия настраивается непосредственно в сценарии сборки. Отпадает необходимость обновлять установленный инструмент, так же как отпадает необходимость предпринимать дополнительные меры по установке Gradle на серверы непрерывной интеграции; для сборки на них также можно использовать Wrapper, загруженный из системы управления версиями. Создатели Gradle рекомендуют использовать Wrapper вместо установки инструментов вручную.

3.2.5. Другие инструменты сборки

В качестве примеров в этой главе демонстрировались инструменты сборки для Java. Однако существует много инструментов для других языков программирования. Одни из наиболее широко используемых перечислены ниже.

- Rake [15] — написан на Ruby и предназначен для сборки программ на этом языке. Следует традициям Make и требует определения правил создания целевых файлов из исходных, например файла со скомпилированным байт-кодом из файла с исходным кодом на Java. Поддерживает расширение новыми возможностями, реализованными на Ruby, поэтому в принципе может использоваться для сборки любых проектов.

- Buildr [16] — использует Rake и дополняет его поддержкой требований, типичных для проектов на Java, позволяя определять задачи, обычные в контексте Java.
- Grunt [17] — инструмент для JavaScript. По сути, это простой инструмент для запуска задач. Однако Grunt в основном используется для выполнения типичных задач, связанных со сборкой проектов на JavaScript, таких как сжатие кода или тестирование. Grunt написан на JavaScript и, соответственно, может расширяться с использованием JavaScript.
- sbt [18] — инструмент для Scala, но также может использоваться для сборки проектов на Java. Так же как в Gradle, определение процедуры сборки записывается на специализированном, предметно-ориентированном языке (DSL), но в данном случае, основанном на языке Scala. Sbt, может осуществлять сборку и тестирование, выполняясь в фоновом режиме, что позволяет быстро получать обратную связь в ситуациях, когда компиляция или тесты перестают работать.
- Leiningen [19] — написан на языке Clojure и для определения процедуры сборки использует предметно-ориентированный язык. Использует декларативный стиль оформления определений, подобно Maven.

3.2.6. Выбор правильного инструмента

Выбор инструмента сборки играет решающую роль, когда дело доходит до реализации непрерывного развертывания. Однако этот выбор обуславливает некий круг проблем, с которыми придется столкнуться. Начиная новый проект, следует заранее определить преимущества выбираемого инструмента и возможность делегирования задач, таких как развертывание, другим инструментам. В принципе такие задачи могут выполняться инструментом сборки, однако применение инструментов, специально предназначенных для развертывания, может оказаться более удачным. В любом случае между различными задачами должны быть определены четкие интерфейсы. В качестве примера можно привести генерацию определенных артефактов, которые затем используются для развертывания.

Обзор инструментов в этой главе показывает, насколько разными могут быть подходы к реализации сборки. Gradle дает бóльшую свободу, не препятствуя использованию соглашений, которые лежат в основе Maven. Но большая степень свободы увеличивает риск хаотизации кода. Поэтому в случае выбора Gradle очень важно регулярно пересматривать и переупорядочивать сценарии сборки. При выборе Maven, напротив, даже простое

расширение и модификация превращаются в очень сложную задачу, если отклоняются от соглашений, принятых в Maven.

Какой бы инструмент вы ни выбрали, всегда следует подробно исследовать имеющиеся возможности и расширения, чтобы узнать, как он может усовершенствовать разработку и сборку.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

1. Установите инструмент командной строки Git (<http://git-scm.com/>).
2. Переведите сборку модуля `user-registration-application` из примера приложения с Maven на Gradle. Для начала извлеките исходный код проекта из GitHub, если вы не сделали этого раньше:

```
> git checkout https://github.com/ewolff/user-registration-V2.git
```

Затем загрузите и установите текущую версию Gradle на свой компьютер. В качестве первого шага перевода сборки в Gradle используйте Build Init [20]. Это позволит быстро сгенерировать первую версию сценария сборки в Gradle:

```
> gradle init
```

Эта команда определит присутствие в текущем каталоге файла модели РОМ для Maven и попытается получить из него всю необходимую информацию о проекте. Для проекта в примере этот подход дает хорошие результаты. Созданный сценарий сборки `build.gradle` для Gradle без проблем скомпилирует проект:

```
> gradle clean build
```

Строку, вызывающую плагин для Maven, можно удалить из файла `build.gradle`. Версия Java теперь должна адаптироваться в соответствии с параметрами `sourceCompatibility` и `targetCompatibility`. Сократите сгенерированный фрагмент `repositories` до единственного вызова `mavenCentral()`, как было показано ранее в этой главе (листинг 3.2). Выполнив новую сборку без удаления файлов, сгенерированных в предыдущей попытке, вы увидите, как происходит инкрементальная сборка. Во время выполнения сборки по завершении каждой задачи будет выводиться сообщение «UP-TO-DATE» (актуально), и Gradle не будет повторно выполнять соответствующие задачи.

На следующем шаге попробуйте встроить в сценарий сборки плагин для Gradle из проекта Spring Boot [21] и установить Gradle Wrapper в проект. Попробуйте также перенести один или несколько модулей из примера проекта в Gradle и объедините сборку двух модулей в одном сценарии [22]. Обратите особое внимание на то, какую информацию можно перенести в общую для двух проектов конфигурацию Gradle (по аналогии с родительской моделью POM в Maven).

3.3. Модульные тесты

Модульные тесты оценивают функциональность небольших фрагментов программного обеспечения — модулей. Обычно они ограничиваются тестированием отдельных методов классов или, максимум, функциональностью целого класса. Зависимости от других классов в модульных тестах игнорируются или подменяются на время тестирования. Соответственно, модульные тесты очень отличаются от интеграционных и приемочных тестов, которые проверяют программное обеспечение целиком и выполняются в полноценном окружении.

К счастью, в настоящее время создание модульных тестов для большинства разработчиков является обычной процедурой. Тем не менее все еще попадают проекты, имеющие крайне низкий охват тестами, — особенно это касается старых проектов. Для успешного внедрения непрерывного развертывания модульные тесты являются обязательным условием. Они образуют в дополнение к тестам других типов, рассматриваемых в последующих главах, первую линию обороны и гарантируют, что изменения в коде не нарушат имеющуюся функциональность. Качество модульного тестирования определяется степенью охвата программного кода. Охват тестами измеряется в процентах числа строк, явно или неявно опробуемых в ходе тестирования, от общего числа строк кода. При высоком уровне охвата модульные тесты могут достаточно надежно гарантировать, что ошибки не просочатся в приложение.

В отличие от других видов тестов, модульные тесты обладают преимуществом быстрого получения обратной связи.

- Они тестируют небольшие, четко очерченные функциональные блоки и поэтому просты и понятны для разработчиков, вновь присоединяю-

щихся к проекту. В какой-то степени они даже могут играть роль документации для разработчиков.

- Они могут выполняться совершенно изолированно, без дополнительных зависимостей. Они не требуют каких-то особенных настроек среды выполнения. То есть модульные тесты могут выполняться на компьютере разработчика или на сервере непрерывной интеграции.
- Благодаря имитации внешних зависимостей они выполняются очень быстро.
- Тесты ограничиваются проверкой конкретных функциональных возможностей. Поэтому прекрасно подходят для получения быстрой обратной связи в контексте конвейера непрерывного развертывания.
- Имитация зависимостей является еще одной причиной, почему модульные тесты оценивают ограниченные блоки кода по отдельности: неудача теста объясняется проблемами, имеющимися только в этом небольшом блоке.

С точки зрения методологии непрерывного развертывания модульные тесты прекрасно подходят на роль основы для всей системы тестирования. Эта идея обсуждается далее в контексте пирамиды тестирования (раздел 4.2). В этом контексте особенно важным аргументом в пользу модульного тестирования является обеспечение надежной и стабильной среды выполнения. Обычно выполнение модульных тестов занимает от десятых долей секунды до нескольких секунд. То есть их очень удобно использовать, чтобы убедиться, что во время разработки не была нарушена нормальная работа существующего кода. Разработчик может просто запускать модульные тесты после каждого изменения. Как автоматизированные тесты на этапе выпуска в конвейере непрерывного развертывания, они позволяют быстро получить обратную связь.

Запуск модульных тестов в ходе обычной процедуры сборки по умолчанию осуществляется обоими инструментами, Maven и Gradle. Это гарантирует выполнение модульных тестов для каждого артефакта.

3.3.1. Создание хороших модульных тестов

Помимо тестирования, модульные тесты играют роль документации для разработчиков. Ясные и четко ограниченные тесты позволяют разработчикам легко понять работу тестируемых классов.

Чтобы обеспечить хорошую удобочитаемость кода тестов, многие коллективы следуют соглашению «настройка-выполнение-проверка» («arrange-act-assert»). Согласно этому принципу, тесты делятся на три части.

- **Настройка (Arrange)** — прежде всего код теста определяет поведение имитаций зависимостей и подготавливает тестовые данные.
- **Выполнение (Act)** — выполняется тестируемая функция.
- **Проверка (Assert)** — в заключение тест оценивает результаты, сравнивая полученные значения с ожидаемыми или определяя факт ожидаемых взаимодействий с зависимостями.

В общих чертах структура тестов напоминает схему «Условие-Операция-Результат» («Given-When-Then»), широко используемую в контексте разработки, основанной на функционировании (Behavior-Driven Development, BDD). В разделе 4.6 приводится пример использования BDD-фреймворка JBehave.

Труд разработчиков можно облегчить не только следованием этой простой последовательности, но также путем присваивания тестовым методам таких имен, которые подсказывали бы, какой вид поведения проверяется тестом. Например, простой тест для примера приложения мог бы выглядеть, как показано в листинге 3.3.

Листинг 3.3. Модульный тест, следующий соглашению «настройка-выполнение-проверка»

```
@RunWith(MockitoJUnitRunner.class)
public class RegistrationServiceUnitTest {

    @Mock
    private JdbcTemplate jdbcTemplateMock;

    @InjectMocks
    private RegistrationService service;

    @Test
    public void registerNewUser() {
        // настройка
        User user = new User(
            "Bastian", "Spanneberg",
            "bastian.spanneberg@codecentric.de");
        // выполнение
        boolean registered = service.register(user);
        // проверка
        assertThat(registered, is(true));
    }
}
```

```
        verify(jdbcTemplateMock).update(
            anyString(),
            eq(user.getFirstname()),
            eq(user.getName()),
            eq(user.getEmail()));
    }
}
```

Однако этот тест использует обычную службу `RegistrationService`. Как правило, для организации доступа к базе данных применяется компонент `JdbcTemplate`, который используется службой `RegistrationService`. Но в этом примере вместо действующего компонента `JdbcTemplate` используется его фиктивная реализация, о чем сообщает аннотация `@Mock`. Благодаря аннотации `@InjectMocks` эта фиктивная реализация внедряется в `RegistrationService`. В самом начале создается тестовый объект, представляющий клиента. Это этап настройки (Arrange). Затем производится регистрация клиента (этап выполнения — Act), после чего проверяется успех регистрации клиента. В заключение в ходе этапа проверки (Assert) также проверяется, произошел ли ожидаемый вызов `JdbcTemplate`.

В этом примере в роли фреймворка тестирования используется Junit [23]; этот фреймворк используется в многочисленных проектах на языке Java. Также широко используется альтернативный фреймворк — TestNG [24]. Для создания имитаций, то есть фиктивных реализаций зависимостей, не являющихся частью тестов, обычно используется фреймворк Mockito [25]. Как можно заметить в примере, при использовании текущей версии Mockito нет необходимости вручную генерировать фиктивные реализации в коде теста; разработчик может просто подставить нужные аннотации. Объект, реализующий тестирование, также отмечается соответствующей аннотацией. Mockito внедряет в этот объект сгенерированные имитации посредством механизма рефлексии.

В большинстве модульных тестов не требуется производить тестовые вызовы сгенерированных имитаций, таких как вызов `JdbcTemplate` в примере выше. Вместо этого достаточно проверить значение, возвращаемое методом. Тест должен как можно меньше знать о внутреннем устройстве тестируемого метода. Реализация теста, представленная в примере, потерпит неудачу, если доступ к базе данных будет осуществляться иначе, чем через `JdbcTemplate`. Поэтому она должна проверить только факт успешной регистрации пользователя. Для этого достаточно проверить значение, возвращаемое методом `register`, или прочитать информацию о новом пользователе из базы данных. В некоторых ситуациях — например, при тестировании

и реорганизации унаследованного кода или при тестировании методов, не имеющих возвращаемого значения, — использование метода `verify()` оправдано. Такой подход позволит оценить правильность работы тестируемой логики.

3.3.2. Разработка через тестирование

В оптимальном случае разработчики пишут тесты до реализации функций. Этот подход известен как «Разработка через тестирование» (Test-Driven Development, TDD).

При использовании методологии TDD разработка, как правило, имеет циклический характер. Сначала создается тест, описывающий желаемое поведение. Поскольку в этот момент реализация еще отсутствует, тест терпит неудачу. Это «красная» стадия. Затем реализуется требуемая функция. Цель состоит в том, чтобы в ходе тестирования получить «зеленую полосу». Далее, по мере необходимости, выполняется рефакторинг теста и прикладного кода. Если в результате этих действий возникает ошибка, соответствующий тест «краснеет», и цикл начинается сначала. Как вы понимаете, тест должен запускаться снова и снова. Для обозначения этого цикла был даже придуман специальный термин: «красный–зеленый–рефакторинг» (рис. 3.2).

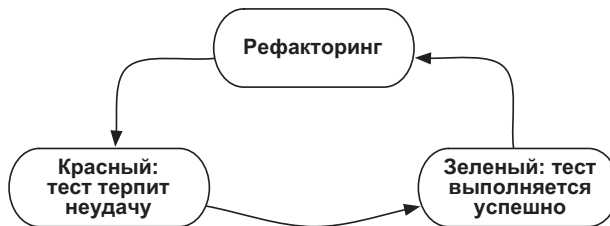


Рис. 3.2. Цикл красный–зеленый–рефакторинг

Первое время этот подход вызывает сложности у многих разработчиков, потому что, например, в момент создания теста они не знают, как будет реализована функция, какие классы будут использоваться и как они должны взаимодействовать. Возникает подспудное желание отложить создание теста до того момента, как вся эта информация станет известна. Однако именно в этом заключается одно из самых больших преимуществ TDD. Эта методология учит разработчиков вносить изменения

мелкими порциями. Поскольку разработчик всегда стремится получить «зеленую полосу», у него нет другого выбора, кроме как двигаться к цели небольшими шагами. В результате данный подход часто оказывается самым простым и продуктивным, потому что разработчикам не приходится сталкиваться с большими сложностями — они просто двигаются вперед небольшими шагами.

Поскольку тесты всегда пишутся до реализации, гарантируется высокий уровень охвата кода тестами и, как следствие, высокая устойчивость к регрессиям, которые также могут проверяться в ходе непрерывной интеграции. Использование плагинов для интегрированной среды разработки, таких как `Infinitest` [26], которые при любом изменении исходного кода немедленно запускают связанные с ним тесты, еще больше увеличивает выгоды от применения методологии TDD. Разработчики освобождаются от необходимости помнить о выполнении тестов. Упомянутый плагин доступен для Eclipse и IntelliJ.

3.3.3. Чистый код и искусство программирования

В последние несколько лет набирают популярность движения Clean Code и Software Craftsmanship (чистый код и искусство программирования). Участники обоих движений интенсивно исследуют вопросы проектирования чистого кода, а также разные подходы к тестированию. Проекты Code Retreats [27] и Coding Dojos позволяют участникам изучать и опробовать TDD и другие методики.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ Соберите несколько своих коллег и потратьте немного времени (час-другой), чтобы провести сеанс TDD Coding Dojo¹. Для этого вам понадобится Kata — небольшая задача, связанная с реализацией некоторой функции. Примером такой задачи может служить String Calculator Kata [28] Роя Ошерова (Roy Osherove).
- ▲ Выполните упражнение группами по два человека за одним компьютером, периодически передавая клавиатуру друг другу. Попробуйте выполнять такие упражнения регулярно, чтобы внедрить

¹ <https://habrahabr.ru/company/scrumtrek/blog/157985/>. — Примеч. пер.

практику применения TDD в своей компании. Вы можете придумать собственные упражнения, используя подход Taking Baby Steps [29].

- ▲ Ознакомьтесь с принципами SOLID [30]. Подумайте, как их применение влияет на тестируемость и удобочитаемость кода и способствуют ли они применению методики TDD.

3.4. Непрерывная интеграция

После реализации основной процедуры сборки необходимо настроить автоматическое ее выполнение при каждом изменении кода или хотя бы вскоре после внесения важных изменений. При этом изменения, добавленные разными разработчиками, должны постоянно объединяться. Данная задача выполняется сервером сборки, который извлекает текущую версию программного обеспечения из системы управления версиями и запускает сборку.

Данный подход — получивший название «непрерывная интеграция» (Continuous Integration, CI), предложенное Кентом Бекон и Мартином Фаулером — помогает намного быстрее обнаруживать ошибки в программном обеспечении, чем если сборка и интеграция программного обеспечения выполняются только по ночам или даже еще реже. Если бизнес-логика распределена по множеству компонентов — возможно, разрабатываемых разными коллективами, — не только сборка и тестирование, но также объединение текущего кода разных компонентов играют важную роль в выявлении несовместимостей на ранних этапах. В настоящее время методология непрерывной интеграции получила широкое распространение в большинстве компаний и коллективов.

Для реализации непрерывной интеграции было создано большое количество серверов сборки. В мире открытого программного обеспечения большой популярностью пользуется сервер непрерывной интеграции Jenkins [31]. Он берет свое начало в проекте Hudson [32]. Также стоит упомянуть сервер непрерывного развертывания GoCD [33], созданный компанией ThoughtWorks. Первоначально GoCD продавался как коммерческий продукт, но с недавних пор был открыт и распространяется бесплатно. Из коммерческих реализаций наиболее широко известны серверы сборки Atlassian Bamboo [34] и TeamCity [35], созданные компанией JetBrains.

Большинство упомянутых инструментов действует локально, в вычислительном центре компании. Конечно, это означает, что сотрудники должны сами выделять время и знать, как использовать и обслуживать эти серверы. Компании, которые не могут позволить себе тратить средства на это, могут воспользоваться облачными услугами, такими как Travis CI [36] и drone.io [37]. В этом случае непрерывная интеграция выполняется в облаке, благодаря чему отпадает необходимость приобретать, устанавливать и сопровождать дополнительное программное обеспечение. Оба продукта имеют бесплатные версии, которые можно использовать для сборки кода, хранящегося в общедоступных репозиториях, например в GitHub. Плата за поддержку закрытых проектов зависит от количества проектов. Jenkins и Bamboo также доступны в виде облачных версий. Услуги Bamboo предлагаются самим создателем.

Обязательным требованием для использования этих предложений является хранение исходного кода в облачных репозиториях, таких как GitHub или Bitbucket, чтобы серверы непрерывной интеграции не испытывали проблем с доступом к ним. Для многих средних и крупных компаний этот вариант не подходит из соображений безопасности данных, поэтому в них предпочтительнее использовать версии, устанавливаемые локально.

3.4.1. Jenkins

Как уже отмечалось, Jenkins является самым широко используемым сервером непрерывной интеграции. Поэтому на нем мы остановимся подробнее. В терминологии Jenkins сборка называется *заданием* (job), которое выполняется в *рабочем пространстве* (workspace). Это каталог в файловой системе, где сохраняются результаты сборки. В стандартной установке Jenkins позволяет создавать произвольные задания сборки, которые объединяют самые разные шаги, и задания Maven, предполагающие использование Maven. Дополнительные типы заданий доступны в виде плагинов. Поддерживается большое количество настроек, определяющих поведение среды и заданий, общих для всех заданий. Они могут изменяться с помощью страниц веб-интерфейса Jenkins.

○ Parametrization (Параметризация)

Позволяет при необходимости определять дополнительные параметры задания. Настраиваемые параметры сборки также можно передавать извне, без необходимости определять их в сценарии сборки или в конфигурации. Эти параметры также могут определяться как переменные.

○ Source Code Management (Инструмент управления исходным кодом)

На этой странице настраивается доступ к репозиторию, откуда извлекается исходный код. Стандартная установка Jenkins поддерживает CVS и SVN (Subversion). Поддержку других инструментов управления версиями, таких как Git, легко можно добавить установкой плагинов. Конкретный перечень параметров настройки, доступных здесь, зависит от используемого репозитория.

○ Build Triggers (Метод запуска сборки)

Определяет моменты, когда должна запускаться сборка.

- В конкретные моменты времени, независимо от наличия изменений в исходном коде.
- При появлении изменений в репозитории.
- После успешного выполнения других заданий сборки.
- После создания SNAPSHOT-зависимостей проекта на том же сервере сборки.

Кроме того, существует возможность протестировать успех интеграции нескольких компонентов. Однако она доступна только для проектов Maven.

○ Build Environment (Окружение сборки)

В этом разделе можно настроить параметры среды сборки. Например, с помощью плагинов настраивается запуск сборки всегда в пустом рабочем пространстве или организуется чтение дополнительных переменных окружения из файлов.

○ Build (Сборка)

Это — главный раздел, поскольку здесь определяются команды сборки. Для заданий Maven это соответствующий вызов со ссылкой, если необходимо, на используемую модель POM. Для произвольных заданий можно определить несколько шагов сборки и, соответственно, использовать несколько инструментов.

○ Post-Build Actions (Действия после сборки)

Здесь определяются действия, которые должны выполняться после сборки. Примером таких действий служат запуск дополнительных заданий, архивирование или тестирование результатов сборки, отправка

извещений по электронной почте в случае появления ошибок во время сборки. С помощью плагинов Jenkins выполняется много самых разных действий.

Расширение с помощью плагинов

Несмотря на поддержку немалого количества функций в стандартной конфигурации, наиболее полный набор преимуществ сервер Jenkins предоставляет только в сочетании с многочисленными плагинами [38], создаваемыми сообществом Jenkins. С их помощью Jenkins может предложить намного больше. Конечно, с увеличением количества установленных плагинов возрастает опасность появления несовместимостей между ними. Поэтому следует тщательно проверять совместимость новых плагинов с уже существующими перед их установкой. Для такого тестирования используется, например, отдельный экземпляр Jenkins. По аналогии с программным кодом и сценариями конфигурации сборки Jenkins регулярно должно проверяться качество используемых плагинов и заданий. В следующих разделах перечисляется много полезных плагинов.

Плагин SCM Sync Configuration

Этот плагин позволяет разработчику синхронизировать конфигурацию сервера Jenkins и все задания с представлениями с системой управления версиями. На данный момент поддерживаются Subversion и Git. В стандартных настройках Jenkins предполагается сохранение конфигурации и всех заданий. Дополнительные файлы в файловую систему Jenkins можно добавлять вручную. Если плагин активен, он предложит ввести комментарий, описывающий изменения в конфигурации, и отправит изменения вместе с этим комментарием в систему управления версиями. В случае потери данных конфигурация Jenkins восстанавливается из системы управления версиями.

Плагин Environment Injector

Плагин Environment Injector позволяет разработчику управлять переменными окружения сборки разными способами и на разных этапах — например, до или после извлечения исходного кода из репозитория или перед сборкой. Кроме того, он позволяет определить механизм запуска сборки (вручную, посредством плагина SCM, пользователем или родительским заданием сборки) в виде переменных окружения. Эта возможность исполь-

зуются, например, в сценариях сборки для запуска конкретных действий, в зависимости от разных условий.

Плагин Job Configuration History

Сложность конфигураций заданий может расти очень быстро, поэтому настоятельно рекомендуется использовать этот плагин. Он сохраняет изменения в конфигурации сборки в виде отдельных версий и позволяет разработчику сравнивать их в пользовательском интерфейсе Jenkins, чтобы следить за изменениями, вносимыми с течением времени. Если в конфигурацию прокрадется ошибка, этот плагин поможет вернуться к последней стабильной конфигурации.

Плагин Clone Workspace SCM

Этот плагин позволяет сохранить в архиве текущее рабочее пространство после сборки и использовать его в других процедурах сборки в качестве начального состояния. Это особенно удобно в сочетании с инструментами, поддерживающими инкрементальную сборку, такими как Gradle, когда последующие этапы основываются на предыдущих результатах. Данное решение помогает увеличить скорость сборки, потому что выполняются только этапы, исходные данные для которых изменились. Кроме того, в окружениях, где происходят частые изменения, этот плагин гарантирует, что все этапы в конвейере сборки будут основываться на одной и той же версии состояния.

Плагин Build Pipeline

Плагин Build Pipeline заслуживает особого упоминания. Как следует из его названия¹, он служит для конструирования и визуализации конвейера сборки. Основой такого представления конвейера служит возможность запуска одних заданий другими. Начиная с первого задания, Jenkins отображает цепочку последующих заданий, в числе которых могут быть, например, приемочные или нагрузочные тесты или процедура развертывания. Кроме того, плагин позволяет вручную запускать последующие этапы как действия нового типа, выполняемые после сборки. То есть имеется возможность выборочно запускать части конвейера вручную. Например, развертывание в окружение оценки качества должно выполняться не всякий раз, когда выполняется сборка. Этот плагин дает возможность координировать работу конвейера непрерывного развертывания (рис. 3.3).

¹ Build pipeline переводится как «конвейер сборки». — *Примеч. пер.*

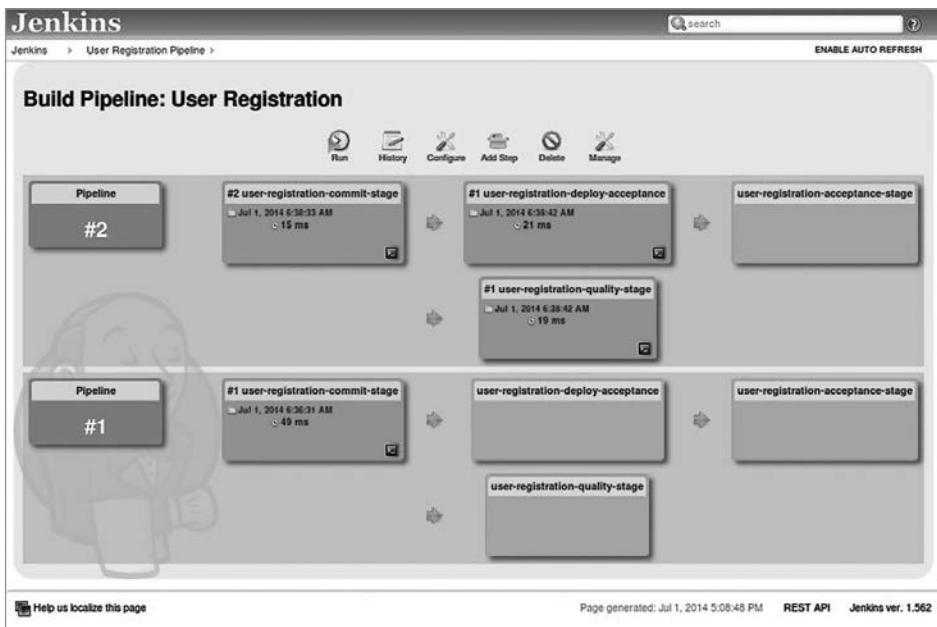


Рис. 3.3. Скриншот с интерфейсом плагина Build Pipeline

Плагин Amazon EC2

Amazon Elastic Compute (EC2) — это облачная служба, предлагающая серверы в аренду. Плагин Amazon EC2 позволяет при необходимости генерировать новые подчиненные узлы Jenkins в облаке EC2 и передавать им задания для сборки. Если узлы не используются некоторое установленное время, плагин автоматически уничтожает их, чтобы снизить затраты на аренду. Конечно, для использования этого плагина необходимо иметь учетную запись в Amazon Web Services (AWS). Если правила, установленные в компании, позволяют это, плагин позволит легко и просто масштабировать Jenkins с увеличением нагрузки, без необходимости расширять имеющуюся в компании инфраструктуру. Кроме того, облачная услуга позволяет обслуживать очень высокие нагрузки.

Плагин Job DSL

С увеличением количества заданий их настройка вручную в веб-интерфейсе Jenkins быстро превращается в сложную работу. Кроме того, нередко имеются задания, логически связанные друг с другом, которые должны изменяться

все вместе. Также на крупных предприятиях постоянно встает вопрос — как максимально просто восстановить задания в случае потери данных (см. также описание плагина SCM Sync Configuration). Плагин Job DSL предлагает решение этих проблем. Как можно заключить из названия¹, плагин предлагает предметно-ориентированный язык (DSL) для создания заданий и представлений в Jenkins. Сценарии с описанием одного или нескольких заданий на этом языке можно поместить в так называемые первоначальные задания вместе со связанными представлениями. Сценарии на предметно-ориентированном языке могут также ссылаться на системы управления версиями. Если выполнить такое задание, оно создаст или обновит все описанные в нем элементы. В листинге 3.4 показан сценарий Job DSL, создающий два взаимозависимых задания и связанное с ними представление конвейера.

Листинг 3.4. Сценарий Job DSL для примера приложения

```
job('user-registration-parent-build') {
    scm {
        git {
            remote {
                url('https://github.com/ewolff/user-registration-V2')
            }
        }
    }
    triggers {
        scm('H/15 * * * *')
    }
    steps {
        maven {
            goals('-e clean install -pl .')
            rootPOM('pom.xml')
            mavenInstallation('Maven 3')
        }
    }
    publishers {
        publishCloneWorkspace('')
    }
}

job('user-registration-build') {
    scm {
        cloneWorkspace('user-registration-parent-build')
    }
    triggers {
        upstream('user-registration-parent-build', 'SUCCESS')
    }
}
```

¹ Job DSL переводится как «предметно-ориентированный язык заданий». — *Примеч. пер.*

```
}
steps {
  maven {
    goals('-e clean install')
    rootPOM('user-registration-application/pom.xml')
    mavenInstallation('Maven 3')
  }
}
publishers {
  publishCloneWorkspace('')
}
}

buildPipelineView('user-registration-pipeline') {
  selectedJob('user-registration-parent-build')
}
```

Как видите, плагин не только поддерживает стандартные возможности Jenkins, но также способен использовать другие плагины, такие как уже упоминавшиеся Clone Workspace и Build Pipeline. Подробное описание программного интерфейса отдельных элементов вы найдете в удобном обзорщике электронной документации Job DSL API Viewer [39].

Создание собственных плагинов

Хотя количество плагинов, разработанных сообществом, огромно и они покрывают большинство потребностей, иногда все же возникает необходимость написать собственный плагин. Jenkins предлагает множество точек входа для разработки расширений. В документации имеется раздел с руководством, описывающим процедуру разработки плагинов [40]. Кроме того, отличной отправной точкой может стать изучение исходного кода имеющегося плагина, чтобы познакомиться с моделью разработки.

3.4.2. Инфраструктура непрерывной интеграции

Если нет возможности пользоваться облачными службами или использовать один из общедоступных серверов непрерывной интеграции, приходится задумываться о создании собственной инфраструктуры непрерывной интеграции и искать ответы на следующие вопросы:

- Как организовать свою инфраструктуру?
- Как поддерживать эту инфраструктуру?
- Какие требования предъявляются к инфраструктуре?

Часто при разработке своего решения непрерывной интеграции эти вопросы не задаются или не рассматриваются, хотя ответы на них крайне важны, особенно в долгосрочной перспективе. Ошибочное решение может повлечь ненужные трудозатраты и привести к неприятному финалу.

К сожалению, часто встречается антишаблон, когда сервер непрерывной интеграции разворачивается на оборудовании, «ставшем ненужным». Обычно это сильно устаревшее оборудование, не обладающее большими вычислительными возможностями, которое не может использоваться ни в каких других целях. Разработчики часто недооценивают важность сервера непрерывной интеграции, который играет ведущую роль в цепочке развертывания программного обеспечения. Отказы и ошибки на этом сервере тормозят весь процесс. Главная задача сервера непрерывной интеграции — как можно быстрее сообщить о корректности или качестве кода. Если его жесткий диск меньше и медленнее, чем жесткие диски на компьютерах разработчиков, или подключение к сети имеет очень низкую пропускную способность, такой сервер легко может превратиться в узкое место и остановить весь проект, вызвав последующие непредвиденные расходы.

Другой вопрос касается структуры и масштабирования инфраструктуры непрерывной интеграции: будет ли использоваться единственный ведущий сервер, раздающий задания подчиненным узлам?

Такая структура (рис. 3.4) предпочтительна тем, что все настройки сосредоточены в одном месте — на ведущем сервере. Это упрощает создание резервных копий конфигураций и заданий. С другой стороны, увеличение количества заданий и пользователей приводит к увеличению нагрузки, что в свою очередь повышает риск сбоя. Когда на ведущем сервере происходит сбой, вся инфраструктура сборки оказывается недоступной. Дополнительные сложности вносит также необходимость администрирования и координации плагинов. Разные команды могут использовать разные наборы плагинов, а это увеличивает риск появления несовместимостей.

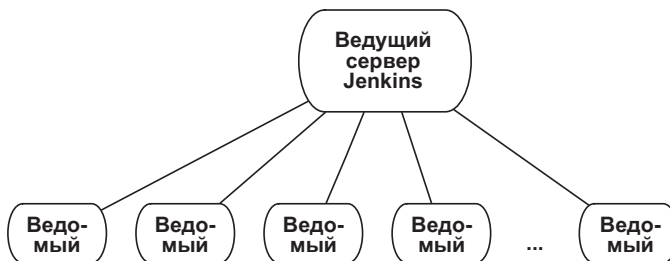


Рис. 3.4. Централизованная инфраструктура непрерывной интеграции

Другой подход заключается в предоставлении каждой команде отдельного ведущего сервера и, если необходимо, дополнительных подчиненных узлов (рис. 3.5).



Рис. 3.5. Децентрализованная инфраструктура непрерывной интеграции

Это делает команды независимыми друг от друга и свободными в определении наборов необходимых им плагинов. С другой стороны, каждый ведущий сервер приходится администрировать и сопровождать отдельно. Кроме того, в этом сценарии исчезает возможность выполнять резервное копирование централизованно.

Вам придется самостоятельно решить, какой вариант предпочтительнее в вашем конкретном случае и с какими проблемами придется столкнуться позже.

3.4.3. В заключение

Создавая инфраструктуру непрерывной интеграции, необходимо учитывать множество факторов. Облачные решения доставляют меньше хлопот на начальном этапе, имеют невысокую стоимость и могут служить хорошей основой для открытых проектов. В комбинации с закрытыми репозиториями они являются отличным выбором для предприятий. Однако для многих компаний передача уязвимых бизнес-данных, таких как исходный программный код, в облачное окружение является совершенно неприемлемым вариантом. В таких случаях инфраструктуру непрерывной интеграции необходимо создавать в пределах компании, и она должна действовать локально. В подобных ситуациях желательно как можно раньше начать думать о будущем и выбрать такую организацию инфраструктуры, которая справится с расширением и изменением требований. В любом случае избегайте решений «на скорую руку». Сервер непрерывной интеграции

представляет первый важный компонент конвейера непрерывного развертывания, и поэтому к его созданию следует подходить с максимальным вниманием и тщательностью.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Настройте задание непрерывной интеграции для проекта примера. Можете использовать для этого виртуальную машину Vagrant, входящую в состав проекта.

1. Внутри проекта перейдите в каталог `ci-setup` [41]. Выполните в нем команду `vagrant up`. В первый раз этой команде может потребоваться несколько минут для выполнения. Когда виртуальная машина запустится, по адресу `http://localhost:9191` будет доступен локальный сервер Jenkins.
2. Чтобы выполнить сборку проекта с помощью Maven, сначала установите версию Maven в Jenkins. Для этого перейдите по ссылке Manage Jenkins («Настроить Jenkins»), находящейся на начальной странице, а затем по ссылке Configure System («Конфигурирование системы»). В разделе Maven выберите версию Maven, которую Jenkins загрузит непосредственно с сайта проекта Apache, если понадобится.
3. Теперь можно заняться фактической настройкой задания сборки. Прежде всего следует определить репозиторий, откуда Jenkins будет загружать исходный код. Прямо сейчас можете использовать адрес `https://github.com/ewolff/user-registration-V2.git`. Если позднее вы захотите внести в проект свои изменения, создайте свою копию репозитория с проектом примера (см. `https://help.github.com/articles/fork-a-repo`). После этого вы сможете вносить изменения в эту копию и запускать сборку в Jenkins. Чтобы создать свою копию, вам понадобится учетная запись GitHub. Так как проект примера основывается на Maven, для его сборки необходимо определить задание Maven. Для этого щелкните на ссылке New Item («Создать элемент») в левом верхнем углу на начальной странице Jenkins и выберите пункт Build a Maven Project («Создать проект Maven»).
4. В разделе Source Code Management («Управление исходным кодом») с настройками прежде всего нужно указать адрес репозитория.

тория, откуда Jenkins должен извлекать код. Для этого выберите систему управления версиями Git и введите URL репозитория, указанный выше. В поле Build Triggers («Метод запуска сборки») укажите, когда Jenkins должен выполнять сборку проекта. Лучший вариант в данном случае — Poll SCM («По опросу системы управления версиями»). В этом случае Jenkins будет проверять наличие изменений в репозитории через регулярные интервалы времени и при их обнаружении — запускать задание. В соответствующих настройках требуется ввести выражение в стиле Cron, определяющее интервал проверки. Например, выражение «*/10 * * *» обеспечит проверку изменений через каждые 10 минут.

5. Наконец, в разделе Build («Сборка») необходимо указать путь к файлу модели Maven POM и цели Maven, которые должны выполняться в ходе сборки. Важным пунктом является clean install («чистая установка»). Если его выбрать, в начале сборки все артефакты, созданные прежде, будут удалены и проект собран заново.
6. Сохраните задание и щелкните на ссылке Build now («Собрать сейчас»), чтобы запустить сборку.
7. Для того же проекта настройте произвольное задание, которое будет служить альтернативой только что созданному заданию Maven. Для этого сначала добавьте в разделе Build («Сборка») новый шаг Invoke top-level Maven targets («Вызвать цели Maven верхнего уровня»), который выполнит сборку проекта Maven. Аналогично встраиваются дополнительные шаги, например вызовы сценариев командной оболочки или сборки целей Ant.
8. В примере сервера Jenkins уже установлен плагин Build Pipeline, упоминавшийся выше. Для дальнейших экспериментов создайте представление конвейера (pipeline view). Для этого перейдите на вкладку «+» на главной странице и выберите только что созданное задание в качестве начальной точки. По мере дальнейшего чтения книги вы сможете дополнить этот конвейер новыми этапами, такими как выполнение нагрузочных тестов, добавляя новые задания, запускаемые начальным заданием сборки.

После знакомства с описанными выше возможностями добавьте задания и представления, генерируемые сценарием Job DSL.

- ▲ Для этого создайте произвольное задание и добавьте в раздел Build («Сборка») шаг Process Job DSLs («Обработать сценарии Job DSL»).

Затем выберите пункт Use the provided DSL script («Использовать имеющиеся сценарии»). После этого вам будет предложено ввести имя сценария в текстовое поле. Чтобы познакомиться поближе с доступными командами, воспользуйтесь обозревателем Job DSL API Viewer, упоминавшимся выше.

- ▲ Поэкспериментируйте также с облачной услугой TravisCI. В ней можно зарегистрироваться со своей учетной записью GitHub. Если вы уже создали копию репозитория с проектом примера в ходе выполнения предыдущих упражнений, проект уже должен быть собран в Travis, поскольку конфигурация TravisCI в файле `.travis.yml` уже присутствует в проекте. Узнайте, какие возможности предлагает Travis, и настройте свой проект для сборки в Travis. На сайте Travis имеются руководства по настройке для разных языков программирования. Руководство для Java находится по адресу: <http://docs.travis-ci.com/user/languages/java/>.

3.5. Оценка качества кода

Мы уже обсудили, как с помощью модульных тестов проверять функциональные возможности небольшими частями. Другие виды тестирования, такие как приемочные испытания и тестирование производительности, обсуждаются в последующих главах. Однако существует еще одна характеристика программного обеспечения, которая также измеряется автоматизированным способом: качество кода.

В большинстве случаев снижение качества программного обеспечения — медленный процесс. А последующие расходы, вызванные снижением качества, накапливаются постепенно. В частности, с течением времени возрастают трудозатраты, необходимые для изменения программного обеспечения, а скорость развертывания новых возможностей уменьшается. Поэтому качество кода тоже нужно непрерывно проверять в конвейере непрерывного развертывания и — как в случае с другими видами тестирования — прерывать работу конвейера, если программное обеспечение не соответствует требованиям. Для оценки качества кода используются параметры.

Например, сложность класса или метода оценивается количеством возможных путей выполнения. Чем больше условных операторов имеется,

тем больше количество возможных путей выполнения и, соответственно, тем сложнее понимать и тестировать код. Увеличение сложности затрудняет сопровождение кода и увеличивает риск появления скрытых ошибок.

Кроме того, в большинстве случаев измеряется также степень охвата кода модульными тестами. Охваченной считается каждая строка кода, выполняемая модульным тестом. В терминах степени охвата кода различают охват строк и охват ветвей (условных инструкций). Под охватом строк подразумевается отношение количества выполненных к общему количеству строк кода. Для оценки степени охвата ветвей требуется определить количество выполненных ветвей. В следующем фрагменте, например, охват строк составит 80%, если во время тестирования переменная `myCondition` получит значение `true`. Однако охват ветвей составит только 50%, поэтому выполнение пойдет только по одному из двух возможных путей.

```
If ( myCondition ) {  
    statement1;  
    statement2;  
    statement3;  
    statement4;  
} else {  
    statement1;  
}
```

Следовательно, чтобы получить надежные оценки охвата кода, всегда нужно рассматривать оба значения.

В дополнение к определению этих классических характеристик также часто добавляют статический анализ кода, чтобы гарантировать его соответствие некоторым определенным правилам. Например, в ходе этого анализа проверяется отсутствие комментариев `// TODO`, следование соглашениям об именовании классов или пакетов или выполнение операций по сохранению трассировочной информации перед повторным возбуждением исключений.

В мире Java существует множество инструментов, помогающих определить эти правила и значения, и оценить степень соответствия им. Наиболее широко в практике используются инструменты статического анализа Checkstyle [42], FindBugs [43] и PMD [44] и средства оценки охвата кода Emma [45], Cobertura [46] и JaCoCo [47]. Эти инструменты могут вызываться из сценариев сборки или с помощью плагинов для сервера непрерывной интеграции.

3.5.1. SonarQube

Итак, чтобы действительно оценить качество программного кода, необходимо установить несколько инструментов и использовать их одновременно. В качестве альтернативы всем инструментам, упомянутым выше, во многих компаниях используется программа SonarQube [48] (прежде известная под названием Sonar) — в первую очередь для проектов на Java, но с течением времени спектр поддерживаемых языков становится все шире. SonarQube оценивает многочисленные характеристики, что и инструменты, перечисленные выше, а также имеет графический интерфейс для управления правилами и оценки результатов (рис. 3.6). Имеется возможность настраивать отдельные панели управления для отдельных проектов. Кроме того, поддержка дополнительных языков, таких как JavaScript или PHP, может быть реализована в SonarQube в виде дополнительных плагинов, аналогично расширяются и функциональные возможности программы.

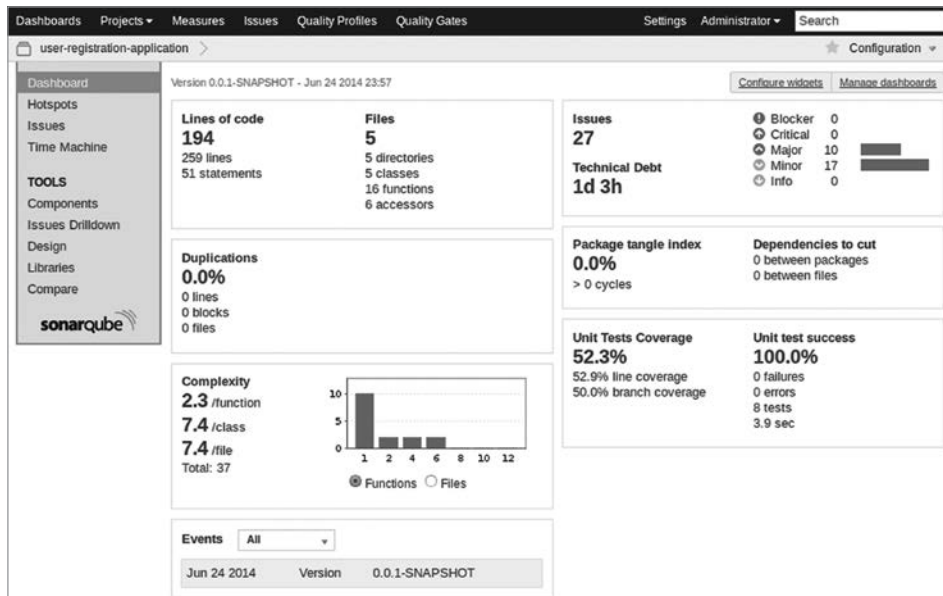


Рис. 3.6. Панель управления для проекта примера в SonarQube

Интеграция в конвейер

SonarQube интегрируется в процедуру сборки, настраивается и вызывается через соответствующие плагины, или ее запуск можно интегрировать через плагин сервера непрерывной интеграции.

Для проектов Maven [49] и Gradle [50] имеются специализированные плагины, существенно упрощающие интеграцию. В обоих случаях необходимо определить набор свойств в файле сборки, чтобы сообщить плагину необходимые учетные данные и адрес экземпляра SonarQube с соответствующей базой данных (листинг 3.5). В листинге 3.5 показано, какие свойства требуется определить для плагина поддержки Maven. Имя пользователя и пароль указываются в файле `settings.xml` с настройками Maven на компьютере, где выполняется анализ, а не в модели РОМ. Файл `settings.xml` содержит настройки для конкретного сервера. Иначе модель РОМ нельзя будет поместить в систему управления версиями, не сообщив всем разработчикам пароль для сервера SonarQube. Дополнительные сведения о настройке плагинов SonarQube для Jenkins можно найти в документации.

Листинг 3.5. Конфигурация SonarQube для проектов Maven

```
<properties>
  <sonar.jdbc.url>
jdbc:mysql://localhost:3306/sonar?autoReconnect=true&use
Unicode=true&characterEncoding=utf8
  </sonar.jdbc.url>
  <sonar.jdbc.username>sonar</sonar.jdbc.username>
  <sonar.jdbc.password>sonar</sonar.jdbc.password>
  <sonar.host.url>http://localhost:9000</sonar.host.url>
</properties>
```

Анализ кода запускается простой командой Maven:

```
> mvn clean install sonar:sonar
```

Плагин извлечет из экземпляра SonarQube настроенные правила, выполнит анализ кода и сохранит результаты в базе данных. Интеграция с процедурой сборки проектов Gradle выполняется аналогично.

Интеграцию можно осуществить не только внедрением команд в сценарий сборки, но также непосредственно на сервере непрерывной интеграции, посредством плагина. В настоящее время доступны плагины для Jenkins и Bamboo. Плагин для Jenkins [51] имеет те же параметры, что описывались выше, они настраиваются централизованно — в меню настройки сервера непрерывной интеграции. Имеется возможность настроить несколько экземпляров, действующих параллельно. Плагин добавляет в Jenkins дополнительное задание сборки SonarQube и действие SonarQube для выполнения после сборки с целью запуска анализа.

Выбор варианта также зависит от других используемых инструментов. Если применяется плагин для сервера непрерывной интеграции, данные и конфигурацию лучше хранить вместе с ним. Кроме того, такой подход позволяет гарантировать постоянную доступность самой актуальной информации о качестве кода. Другой подход — интеграция в сценарий сборки — тоже возможен, но возникает некоторая сложность из-за того, что в наши дни каждый проект может иметь свой сервер непрерывной интеграции. В любом случае, включая интеграцию в сценарий сборки, важно обеспечить доступность данных, хранящихся за пределами системы управления версиями в защищенном хранилище.

Так или иначе, использование SonarQube или других подобных инструментов для непрерывной оценки качества кода является решающим фактором, позволяющим увеличить скорость разработки новых особенностей.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ Поэкспериментируйте с разными способами интеграции SonarQube в свой конвейер сборки. На виртуальной машине Vagrant с сервером непрерывной интеграции, входящей в состав проекта примера, уже установлен сервер SonarQube. После успешного запуска он становится доступным по адресу <http://localhost:9393>. На сервере Jenkins плагин для интеграции с SonarQube уже установлен. Сервер Jenkins доступен по адресу <http://localhost:9191>. С учетными данными «admin/admin» вы можете зарегистрироваться на сервере SonarQube как администратор и в разделе Quality Profiles («Профили качества») изменить имеющийся набор правил или создать новый.
- ▲ Попробуйте сначала интегрировать анализ в сценарий сборки (Maven или Gradle) или запускать его из своего задания на сервере Jenkins.
- ▲ Затем настройте интеграцию через плагин для Jenkins. Укажите учетные данные для доступа к своему серверу SonarQube в Jenkins, в разделе Manage Jenkins > Configure System («Настроить Jenkins > Конфигурирование системы»). После этого вы сможете выполнять анализ в ходе сборки или отдельно, как отдельное задание в Jenkins. Такой подход позволяет выполнять анализ параллельно с приемочными тестами, чтобы сократить время получения обратной связи.

- ▲ Чтобы еще больше увеличить ценность SonarQube, анализ должен прерываться с ошибкой при превышении некоторого порога. Для определения подобных порогов SonarQube предлагает объявлять так называемые *ворота качества* (quality gates). Чтобы при этом также прерывать работу соответствующего задания сборки, необходимо установить плагин Build Breaker для SonarQube. Для этого зарегистрируйтесь как администратор (admin/admin) и перейдите по ссылке Settings («Настройки») в правом верхнем углу. В открывшемся меню выберите пункт для перехода в центр обновления (update center), где перейдите на вкладку Available Plug-Ins («Доступные плагины»). Здесь вы сможете выбрать и установить плагин Build Breaker. После этого свяжите свой проект с воротами качества в разделе Quality Gates («Ворота качества»). Для начала можно использовать уже готовые ворота качества SonarQube way. Позднее вы сможете подобрать значения для своих ворот качества, более подходящие для текущего проекта.

3.6. Управление артефактами

Еще один важный аспект, который не следует недооценивать и который оказывает существенное влияние на планомерную разработку и бесперебойное функционирование конвейера непрерывного развертывания, — администрирование и обслуживание сгенерированных артефактов. Благодаря широкому использованию Maven появился термин «репозиторий артефактов». Эти репозитории играют важную роль в Maven. Зависимости загружаются из репозитория, а созданные артефакты выгружаются в репозиторий. Репозиторий определяется в модели POM. Для этих целей существуют общедоступные репозитории. Однако также можно организовать локальные репозитории. Репозитории, создаваемые локально, внутри компании, хранят результаты сборки локальных проектов и часто играют роль буферов для загрузки внешних артефактов из общедоступных репозитория. Такой подход позволяет сделать процесс сборки независимым от доступности внешних репозитория, потому что нет никаких гарантий, что общедоступные репозитории продолжают хранить библиотеки нужных версий через несколько лет после их выпуска. Кроме того, можно также определить, какие внешние источники вообще могут использоваться. То есть внутренние репозитории играют центральную роль

в управлении внутренними и внешними артефактами. То же верно в отношении разработчиков и централизованного процесса сборки.

Для конвейера непрерывного развертывания репозиторий артефактов играет особенно важную роль, потому что он служит главным хранилищем всех артефактов, необходимых в процессе развертывания.

Как показано на рис. 3.7, репозиторий артефактов является основой для всех действий, выполняемых в конвейере. На этапе выпуска генерируются артефакты, подлежащие развертыванию. При этом через репозитории промежуточных или выпускаемых версий удовлетворяются зависимости от других внутренних проектов. Зависимости от внешних проектов удовлетворяются через прокси-репозитории в ходе сборки. В ходе сборки также исследуются все критерии оценки качества: успешное выполнение тестов, охват кода тестированием и соответствие кода принятым правилам и соглашениям. Далее артефакты помещаются в репозиторий.

Последующие шаги, такие как развертывание в испытательном окружении и, наконец, в рабочем окружении или в окружении для обкатки, используют артефакты, находящиеся в репозитории. То есть репозиторий артефактов служит интерфейсом между разными этапами процесса.

Чтобы также отобразить в репозитории, как программное обеспечение движется по конвейеру, используются метаданные версии или организуются несколько репозиториев, представляющих отдельные этапы конвейера непрерывного развертывания. В этом случае программное обеспечение передается в новый репозиторий после успешного тестирования. Такой порядок действий называют также конвейеризацией (staging) [52]. Однако обычно эта возможность поддерживается только коммерческими версиями репозиториев. Но, по крайней мере, для Jenkins и сервера репозиториев Nexus существует плагин Artifact Promotion [53], реализующий конвейеризацию путем передачи в другие репозитории.

Артефактами могут быть файлы JAR, WAR или EAR или файлы, более близкие к операционной системе, такие как пакеты Debian или RPM, которые также могут создаваться в конвейере. Многие популярные серверы репозиториев поддерживают артефакты таких типов, правда, некоторые из них только в коммерческих версиях. Преимущество таких артефактов в том, что они более знакомы системным администраторам и легко интегрируются в автоматизированный процесс обновления. Также — и особенно для таких артефактов — существуют альтернативные решения, в которых комплекты сложных настроек для Linux могут распространяться в виде пакетов Debian или RPM.

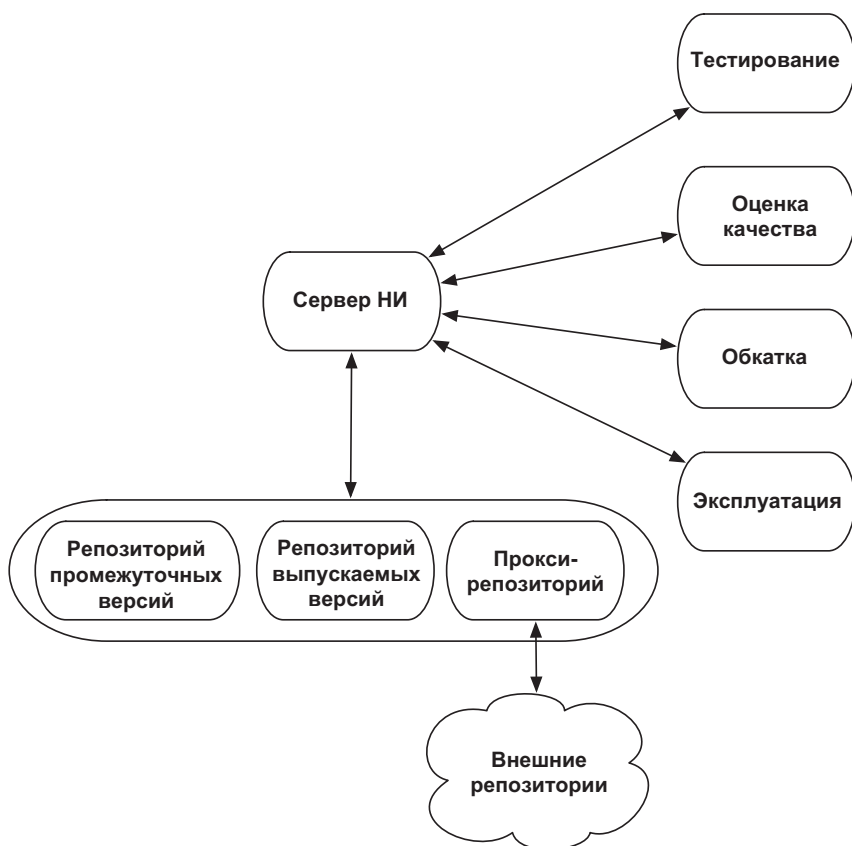


Рис. 3.7. Репозиторий является основой инфраструктуры развертывания

Не имеет значения, координируется ли работа на всех этапах с помощью единого инструмента, такого как сервер непрерывной интеграции на рис. 3.7, или конвейер включает несколько инструментов. Кроме того, коммерческие решения в области автоматизации выпуска и развертывания предлагают адаптеры для организации взаимодействий с наиболее распространенными репозиториями артефактов.

3.6.1. Интеграция в процедуру сборки

Интеграция репозитория в процедуру сборки программного обеспечения легко осуществляется с помощью Maven или Gradle. В обоих случаях поддерживаются два разных типа репозитория.

- Один репозиторий служит для удовлетворения зависимостей — он хранит библиотеки из внешних источников и результаты сборки всех внутренних проектов.
- Другой используется для сохранения сгенерированных артефактов — результатов сборки.

Листинг 3.6. Определение репозитория в Maven POM

```
...
<repositories>
  <repository>
    <id>spring-milestones</id>
    <url>http://repo.springsource.org/milestone</url>
  </repository>
</repositories>
...
<distributionManagement>
  <repository>
    <id>internal-release</id>
    <name>Internal Release Repository</name>
    <url>http://localhost:9292/artifactory/libs-release-local</url>
  </repository>
</distributionManagement>
```

С настройками в листинге 3.6 команда

```
> mvn deploy
```

выполнит сборку артефакта, сгенерирует Java-архивы (файлы JAR) и выгрузит их в репозиторий, настроенный в разделе `<distributionManagement>`. Зачастую также приходится настраивать учетные данные, потому что не каждый должен иметь право выгружать артефакты. Учетные данные принято хранить не в модели POM проекта, а в централизованном месте, например в файле `settings.xml` с настройками Maven, на сервере непрерывной интеграции. Подробное описание настройки учетных данных вы найдете в документации к Maven.

Gradle заимствовал многие идеи из Maven, в том числе и подходы к поддержке репозитория, поэтому аналогичные конфигурации выглядят похожими, как показано в листинге 3.7.

Листинг 3.7. Определение репозитория в Gradle

```
repositories {
  mavenLocal()
  mavenCentral()
```

```
maven { url 'http://repo.springsource.org/milestone' }
}
...
uploadArchives {
    repositories {
        maven {
            url = 'http:// 192.168.33.22:8081/artifactory/libs-release-local'
        }
    }
}
```

Следующая команда Gradle выгрузит созданные архивы в настроенный репозиторий:

```
> gradle uploadArchives
```

Gradle поддерживает не только репозитории Maven, но также более старые репозитории Ivy и простые сетевые диски и другие подобные устройства. Это обеспечивает больше гибкости с точки зрения управления репозиториями, чем Maven. В Maven неявно настраиваются локальный репозиторий Maven и центральный общедоступный репозиторий Maven Central. В Gradle они должны определяться явно, потому что вместо них в принципе могут использоваться другие репозитории.

3.6.2. Дополнительные особенности репозитория

Популярные реализации репозитория, такие как Artifactory [54] и Nexus [55], предлагают намного больше возможностей, чем просто хранение и извлечение артефактов Java, в том числе:

- интерфейс REST для извлечения данных и управления ими;
- развитую систему прав и привилегий пользователей;
- возможность исследовать хранящиеся артефакты с учетом лицензирования.

Часто в одном репозитории можно хранить артефакты других типов, такие как модули Ruby или пакеты Debian и RPM. Поэтому они представляют определенный интерес для проектов на нескольких языках программирования и предлагают возможность централизованно управлять всеми артефактами, сгенерированными в компании, независимо от их типа и назначения, будь то пакеты со сценариями или артефакты, полученные в ходе разработки программного обеспечения. Все эти артефакты можно распределить по соответствующим системам.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ В виртуальной машине Vagrant с настроенным сервером непрерывной интеграции также установлен репозиторий артефактов Artifactory. После успешного запуска он становится доступен по адресу <http://localhost:9292>. Добавьте в сценарий сборки проекта примера сохранение созданных артефактов в этот репозиторий. Используйте в качестве отправной точки фрагменты кода, представленные выше в тексте. Поэкспериментируйте с определением разных учетных записей для загрузки артефактов. Для этого зарегистрируйтесь на сервере с правами администратора `admin/password` и настройте необходимые учетные записи. Сохраните учетные данные на сервере непрерывной интеграции для использования с Maven.
- ▲ Узнайте, имеется ли в вашей компании репозиторий артефактов. Если имеется, кто им пользуется? Только разработчики или он также доступен оперативному персоналу? Распространяются ли через этот репозиторий инструкции по использованию артефактов?
- ▲ Экспериментируя со своим проектом, например, разверните его на сервере приложений, попробуйте интегрировать Artifactory и Jenkins. Для этого можно использовать, например, плагин Repository Connector [56], позволяющий загружать файлы WAR или EAR из репозитория в задание развертывания на сервере Jenkins и затем передавать их — например, с помощью плагина SCP [57] — на сервер, где они должны выполняться.

3.7. В заключение

Прочитав эту главу, вы получили более полное представление об аспектах разработки, сборки и непрерывной интеграции, играющих важную роль в контексте непрерывного развертывания. Выбор инструментов, приемов и методов оказывает существенное влияние на подготовку надежной основы для упрощения последующей автоматизации.

В частности, теперь вам должно быть понятно, что — в точности как программный код — код логики сборки и интеграции должен находиться под неусыпным контролем и своевременно изменяться и дополняться, когда он перестает соответствовать текущим требованиям. В этом контексте также

может всплыть проблема поддержки унаследованного кода. Она становится особенно неприятной, если в худшем случае вся цепочка развертывания программного обеспечения останавливается и все группы оказываются не в состоянии продолжить работу.

Важно понимать, что знание особенностей настройки сборки и непрерывной интеграции не должно ограничиваться единственным человеком или узким кругом лиц. Каждый разработчик должен понимать, что происходит в ходе сборки, и помнить об этом во время разработки. Инструменты, такие как SonarQube, очень пригодятся всем членам команды и добавляют ясности, если используются планомерно и четко запланированным способом и доступны всем членам команды.

В любом случае усилия, предпринятые в этой области, закладывают основу для успешно функционирующего конвейера непрерывного развертывания. Поэтому качество, простота обслуживания и регулярный уход за логикой сборки и связанной с ней инфраструктурой являются важными аспектами.

Ссылки

1. <http://ant.apache.org/>
2. <http://maven.apache.org/>
3. <https://gradle.org/>
4. <http://ant-contrib.sourceforge.net/tasks/tasks/index.html>
5. <http://ant.apache.org/ivy/>
6. <http://maven.apache.org/components/plugins/maven-jar-plugin/>
7. <http://maven.apache.org/>
8. <http://maven.apache.org/maven-release/maven-release-plugin/>
9. <http://www.mojohaus.org/versions-maven-plugin/>
10. <http://maven.apache.org/scm/maven-scm-plugin/>
11. <https://axelfontaine.com/blog/maven-releases-steroids.html>
12. <https://axelfontaine.com/blog/final-nail.html>
13. <https://gradle.org/>
14. <http://www.groovy-lang.org/>

15. <https://github.com/ruby/rake>
16. <http://buildr.apache.org/>
17. <http://gruntjs.com/>
18. <http://www.scala-sbt.org/>
19. <https://leiningen.org/>
20. https://docs.gradle.org/current/userguide/build_init_plugin.html
21. <http://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins-gradle-plugin.html>
22. https://docs.gradle.org/current/userguide/multi_project_builds.html
23. <http://junit.org/junit4/>
24. <http://testng.org/doc/index.html>
25. <http://site.mockito.org/>
26. <http://infinittest.github.io/>
27. <http://coderetreat.org/>
28. <http://osherove.com/tdd-kata-1>
29. <http://blog.adrianbolboaca.ro/2013/03/taking-baby-steps/>
30. [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))¹
31. <https://jenkins.io/index.html>
32. <http://hudson-ci.org/>
33. <https://www.gocd.io/>
34. <https://www.atlassian.com/software/bamboo>
35. <http://www.jetbrains.com/teamcity/>
36. <https://travis-ci.org/>
37. <https://github.com/drone/drone>
38. <https://wiki.jenkins-ci.org/display/JENKINS/Plugins>
39. <http://jenkinsci.github.io/job-dsl-plugin/>
40. <https://wiki.jenkins-ci.org/display/JENKINS/Plugin+tutorial>

¹ [https://ru.wikipedia.org/wiki/SOLID_\(объектно-ориентированное_программирование\)](https://ru.wikipedia.org/wiki/SOLID_(объектно-ориентированное_программирование)). — Примеч. пер.

41. <https://github.com/ewolff/user-registration-V2/tree/master/ci-setup>
42. <http://checkstyle.sourceforge.net/>
43. <http://findbugs.sourceforge.net/>
44. <https://pmd.github.io/>
45. <http://emma.sourceforge.net/>
46. <http://cobertura.github.io/cobertura/>
47. <http://www.eclemma.org/jacoco/>
48. <https://www.sonarqube.org/>
49. <https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+maven>
50. <https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner+for+Gradle>
51. <https://docs.sonarqube.org/display/SONAR/Analyzing+with+SonarQube+Scanner+for+Jenkins>
52. <https://www.jfrog.com/confluence/display/RTF/Build+Integration#BuildIntegration-ReleaseManagement>
53. <https://github.com/jenkinsci/artifact-promotion-plugin/tree/master>
54. <https://www.jfrog.com/artifactory/>
55. <http://www.sonatype.org/nexus/>
56. <https://wiki.jenkins-ci.org/display/JENKINS/Repository+Connector+Plugin>
57. <https://wiki.jenkins-ci.org/display/JENKINS/SCP+plugin>

4

Приемочные тесты

4.1. Введение

Автоматизация приемочных испытаний (acceptance tests) является неотъемлемым требованием, потому что по их результатам можно судить, насколько точно реализованы требования. Однако приемочные тесты — лишь одна из разновидностей тестов. В разделе 4.2 обсуждается пирамида тестирования. Эта пирамида демонстрирует, какие тесты должны присутствовать и какое место они занимают в проекте. В разделе 4.3 детально описываются преимущества приемочных испытаний. Один из подходов к реализации приемочных испытаний заключается в имитации действий пользователя с графическим интерфейсом (ГИП) для проверки совпадения фактического поведения системы с ожидаемым. Этот подход рассматривается в разделе 4.4. Тесты основаны на применении инструмента Selenium. Знакомство с альтернативными инструментами станет темой раздела 4.5.

В разделе 4.6 описывается подход, в котором требования записываются на простом естественном языке. То есть они выглядят как самые обычные требования, но могут проверяться автоматически. Для этого используется Java-фреймворк JBehave [1]. Альтернативные технологии описываются в разделе 4.7. Раздел 4.8 посвящен стратегиям успешной автоматизации приемочных испытаний в проектах. Наконец, в разделе 4.9 подводятся заключительные итоги.

4.1.1. Приемочные испытания: пример

В этой главе описывается одна из особенностей приемочных испытаний — их автоматизация. Автоматизация помогает уменьшить трудозатраты на выполнение тестов. Компания «Big Money Online Commerce Inc.», пред-

ставленная в разделе П.2, также приняла на вооружение эту методику непрерывного развертывания.

Без автоматизации приемочных испытаний в рабочее окружение могут проникать ошибки, которые легко заметить во время тестирования. Как описывалось в разделе П.2, тестирование не было выполнено в компании после исправления ошибки, потому что специалисты решили, что это требует слишком много усилий, поэтому ошибка просочилась в рабочее окружение незамеченной.

Кроме того, автоматизированные тесты помогают воспроизводить одни и те же результаты. В прежние времена проблемы нередко возникали просто потому, что результаты тестирования не воспроизводились, или потому, что люди, выполняющие тестирование, допускали ошибки. Оба этих источника проблем устраняются автоматизацией приемочных испытаний.

И наконец, после начальных вложений в автоматизацию испытаний последующие затраты на тестирование существенно снижаются. Вложения окупаются сторицей. Теперь тесты позволяют разрабатывать и выпускать новые особенности с меньшими усилиями.

4.2. Пирамида тестирования

В главе 3, «Автоматизация сборки и непрерывная интеграция», обсуждались модульные тесты. В этой главе мы рассмотрим способы автоматизации приемочных тестов. Глава 6, «Исследовательское тестирование», покажет, какое место в общей картине занимают исследовательские тесты, выполняемые вручную. Возникает законный вопрос: какое внимание должно уделяться разным видам испытаний. Ответ на этот вопрос поможет получить пирамида тестирования [2] (рис. 4.1). Размер разных разделов пирамиды примерно соответствует относительному количеству тестов различного вида.

В основе пирамиды находятся модульные тесты — тесты этого вида самые многочисленные. Они легко реализуются, не имеют зависимостей за пределами тестируемого класса и выполняются относительно быстро. Поэтому это первоочередной вид испытаний.

Тестирование через программный интерфейс — например, приемочные испытания с применением инструментов разработки, основанной на функ-



Рис. 4.1. Пирамида тестирования

ционировании (Behavior-Driven Development, BDD), о которых рассказывается в этой главе, — представляет следующий уровень. За ним следуют приемочные испытания, выполняемые посредством автоматизированных операций с графическим интерфейсом пользователя (ГИП). Тестирование через программный интерфейс предпочтительнее; эти тесты выполняются быстрее, чем тесты ГИП, потому что не используют слой ГИП и могут выполняться существенно эффективнее. Кроме того, изменения в ГИП не нарушают работу этих тестов. Тестирование через ГИП может терпеть неудачу просто потому, что элемент графического интерфейса был переименован или в интерфейс были внесены какие-то другие мелкие изменения.

Тестирование через ГИП или программный интерфейс позволяет оценить функциональные возможности — в конце концов, они определяют требования к системе. Как следствие, они являются важным дополнением к модульным тестам.

Однако если приемочные тесты через ГИП или программный интерфейс обнаруживают ошибку, требуется написать модульный тест, имитирующий ошибочную ситуацию, потому что такие ошибки должны обнаруживаться уже на стадии модульного тестирования.

Ручное тестирование, находящееся на вершине пирамиды, применяется только в отдельных сценариях. Испытания этого вида трудоемки и трудно воспроизводимы. Если конвейер приходится запускать очень часто, ручное тестирование становится невозможным.

Однако построить свою хорошо сбалансированную пирамиду тестирования в действительности довольно сложно, чаще можно наблюдать противоположную ситуацию: с большим объемом ручного тестирования (рис. 4.2). В этом случае пирамида тестирования превращается в рожок мороженого. Автоматизация ограничивается тестированием взаимодействий через ГИП. Имеется большое количество ГИП-тестов, потому что они легко реализуются на основе тестирования вручную. Количество тестов через программный интерфейс невелико, а если вообще они присутствуют, то используются только в особых случаях. Наконец, имеется некоторое количество модульных тестов. Обычно проекты реализуют исчерпывающий набор модульных тестов. Однако большинство из них имеют крайне низкое качество. Упор на ручное тестирование требует больших затрат времени и, соответственно, расходов. Даже автоматизированные тесты через графический интерфейс выполняются медленно, к тому же они легко повреждаются малейшими модификациями в графическом интерфейсе — простое изменение имени элемента интерфейса может сделать тест неработоспособным. Организация тестирования в форме рожка мороженого — самая частая проблема, касающаяся стратегии тестирования.



Рис. 4.2. В действительности организация тестирования больше напоминает рожок мороженого

Поэтому разработчики всегда должны стремиться получить пирамиду тестирования, изображенную на рис. 4.1, независимо от того, собираются они внедрять методологию непрерывного развертывания или нет. Это — универсальный подход.

Цель методологии непрерывного развертывания — получение обратной связи максимально быстро. Поэтому кроме выявления стратегии тестирования, соответствующей идее пирамиды тестирования, важную роль играет также скорость обнаружения ошибок. То есть тесты должны сначала проверять основные функциональные возможности — простые бизнес-процессы, такие как создание учетной записи клиента или нового заказа. Начальные тесты должны охватывать много процессов: главная их цель состоит в том, чтобы проверить все типичные простые ситуации. Только затем можно переходить к более глубинному тестированию, то есть писать тесты, оценивающие крайние случаи и проверяющие возможные ошибочные ситуации.

Поэтому тесты должны структурироваться так, чтобы сначала они охватывали основные и простые бизнес-процессы и только потом проверялись более сложные ситуации. Если программа содержит критическую ошибку, она должна выявляться быстро, по крайней мере, если она сокрыта в основных функциях программы. Если ошибка будет выявлена в ходе первичного, поверхностного тестирования, более глубинное тестирование не потребуется. Если работоспособность нарушена уже в простых случаях, в более сложных сценариях она будет нарушена тем более. Если каждую особенность сразу тестировать на всю глубину, может так получиться, что критическая ошибка будет выявлена только ближе к концу тестирования. Если ошибка находится в последней тестируемой функции, все остальные особенности к этому моменту будут протестированы на всю глубину, а это немалое время. Разумеется, это вносит задержку в получение обратной связи.

Поэтому иногда имеет смысл делить тесты не по видам — приемочные или нагрузочные, — а по вероятности выявления ошибок и сначала выполнять более обширные и поверхностные, а лишь затем более узкоспециализированные и глубокие. Кроме того, в первую очередь в тестах следует проверять основной поток выполнения бизнес-процесса с простыми параметрами. Крайние случаи и сложные сценарии должны исследоваться позднее.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Проанализируйте стратегию тестирования известного вам проекта.

- ▲ Какая доля текущего набора приходится на исследовательские тесты, то есть исследующие новые особенности?
- ▲ Какая доля текущего набора тестов автоматизирована? В оценке должны участвовать модульные тесты, тесты пропускной способ-

ности, тесты для проверки других нефункциональных требований (удобство использования, безопасность) и функциональные приемочные тесты.

- ▲ Какую фигуру напоминает используемая стратегия тестирования — пирамиду или рожок мороженого?
- ▲ Как проект защищен от регрессий — то есть от ошибок, которые в принципе были устранены, но появились вновь из-за изменений в коде? Автоматизированы ли регрессионные тесты? Какие преимущества дает автоматизация?
- ▲ Тестируются ли нефункциональные требования, такие как удобство использования или «привлекательный внешний вид»?
- ▲ Где больше всего обнаруживается ошибок? Выявляются ли они одним из тестов? Каким? Или они обнаруживаются только после развертывания в рабочем окружении? Хорошим источником этой информации послужит система отслеживания ошибок (bug tracker).

Опираясь на эту информацию, можно разработать план добавления дополнительных тестов и определить очередность их автоматизации.

4.3. Что такое «приемочные тесты»?

Цель приемочных тестов легко угадывается из названия: пользователи или клиенты «принимают» программное обеспечение. Отсюда вытекает важная особенность этих тестов: тестировщики, инженеры-технологи, определяющие требования, разработчики и, наконец, клиенты должны понимать приемочные тесты и убедиться, что тесты действительно охватывают все приемочные критерии. Это требует взаимодействия всех вовлеченных сторон — поэтому приемочные испытания активизируют обмен информацией в пределах проекта.

4.3.1. Автоматизированные приемочные испытания

Для поддержки непрерывного развертывания приемочные испытания должны быть автоматизированы. Если предполагается часто развертывать новые версии и, соответственно, часто выполнять тестирование, без авто-

матизации цена тестирования оказывается слишком высокой. Кроме того, результаты ручных испытаний трудно воспроизводимы, потому что ошибки, обнаруженные в таких испытаниях, необязательно являются следствием ошибок в программном обеспечении, а могут быть вызваны ошибками, допущенными в ходе тестирования.

4.3.2. Больше, чем просто увеличение эффективности

Автоматизация не только повышает эффективность тестирования: она позволяет выполнять тестирование чаще и быстрее получать обратную связь. Когда приемочные испытания выполняются с каждой передачей изменений в репозиторий, разработчики уже в течение нескольких часов или даже минут смогут узнать о наличии ошибок в программном обеспечении. Соответственно, область поиска ошибок может быть ограничена изменениями, внесенными в течение последних нескольких часов.

Однако тестирование вручную выполняется незадолго перед выпуском новой версии — то есть на самых последних этапах, когда намного сложнее выяснить, какие изменения вызвали ошибку. Кроме того, испытания выполняются на этапе, когда группа разработчиков уже находится под давлением приближающегося момента выпуска. Поэтому автоматизация может облегчить процесс устранения ошибок и улучшить качество программного обеспечения.

Усилия, необходимые для автоматизации, часто завышаются: испытания, выполняемые вручную, нередко почти полностью формализованы, но не совсем. Если бы они были формализованы полностью, автоматизировать их было бы очень просто. Например, файл в формате Excel может содержать описание действий, необходимых для получения определенного результата, как в следующем списке.

- Введите следующий текст в поле ввода: «Проверка клиента».
- Нажмите кнопку ОК.
- Проверьте совпадение фактического результата с ожидаемым.

С практической точки зрения, тесты этого типа считаются худшими из возможных. Такой тест сложно сгенерировать, потому что он слишком подробный. Он не просто описывает ожидаемое поведение, но также предполагает использование определенных элементов графического интерфейса. Тестирование чересчур трудоемко, потому что все еще должен быть человек,

выполняющий взаимодействия и проверяющий результаты. В результате тестировщик, понимающий предметную область применения приложения, низводится до уровня автомата. Подобное описание едва ли поможет ему понять проверяемый аспект предметной области. Да, здесь проверяются конкретные значения — но какую ценность имеют эти значения с предметной точки зрения? Какие особые случаи проверяются прямо сейчас? Так как глубокий смысл теста остается неясным, отчет тестировщика по результатам испытаний выглядит таким же неясным, как, например, «что-то пошло не так». По такому отзыву сложно определить, что за проблема скрывается под этим тестом.

Раздел 4.4 демонстрирует, что для автоматизации таких тестов требуется не так много дополнительных усилий. Но автоматизация значительно снижает трудозатраты на выполнение тестов. В то же время тестировщики могут сосредоточиться на задачах, где они с большей пользой смогут применить свои знания предметной области. Как следствие, автоматизация приемочных испытаний приобретает особый смысл с практической точки зрения и быстро окупается.

4.3.3. Тестирование вручную

Методология непрерывного развертывания не требует полной автоматизации всех приемочных испытаний. Этим она отличается от методологии экстремального программирования, согласно которой каждое требование должно поддерживаться автоматизированным тестом. Конвейер непрерывного развертывания включает этап исследовательского тестирования вручную (см. главу 6). Автоматизированные приемочные испытания имеют целью помочь избежать регрессий — ошибок, которые прежде уже были устранены, но появились вновь из-за изменений в коде. Исследовательские тесты в первую очередь предназначены для поиска ошибок в новых модулях и для проверки таких характеристик, как, например, удобство в использовании, измерение которых трудно автоматизировать. Кроме того, группа тестировщиков не обременена необходимостью проводить уже автоматизированные приемочные испытания. То есть ее члены могут сосредоточиться на исследовательских испытаниях, а не проводить тестирование одних и тех же компонентов снова и снова.

4.3.4. Какую роль играет заказчик?

Вообще говоря, приемочные испытания гарантируют приемку программного обеспечения заказчиком. Когда заказчик передает разработку внешне-

му подрядчику, приемочные испытания приобретают особую важность, потому что только после успешного прохождения этих испытаний заказчик оплатит труд подрядчика. Группе разработчиков не всегда удастся убедить заказчика в том, что автоматизированные приемочные испытания действительно отражают выдвинутые им требования. Это подчеркивает особую важность приемочных испытаний для взаимодействий. Когда представители заказчика сами смогут понять и исследовать автоматизированные приемочные тесты, они, вероятнее всего, согласятся с ним как с одним из критериев приемки.

Ничто не мешает объединить формальную приемку с ручными приемочными испытаниями. Однако такое тестирование может выполняться только один раз, а именно — в процессе приемки новой версии заказчиком. Автоматизированные приемочные тесты при этом не теряют своей важности, потому что гарантируют правильное функционирование системы в процессе прохождения через конвейер непрерывного развертывания. Приемочные испытания, проводимые заказчиком вручную, также могут служить основой для автоматизированных приемочных тестов. Ошибки, выявленные заказчиком на этапе приемочного тестирования, должны повлечь создание новых автоматизированных приемочных тестов, гарантирующих фактическое устранение ошибок и невозможность их повторного появления в будущем.

4.3.5. Приемочные и модульные тесты

Наконец, важно различать модульные и приемочные тесты. Главное различие между ними заключается в используемых технологиях. Приемочные тесты также могут быть реализованы с привлечением фреймворков модульного тестирования, таких как JUnit.

Во-первых, самое важное различие между этими двумя типами тестов — разный круг заинтересованных лиц. Модульные тесты — это инструмент разработчика. Они пишутся разработчиками с целью гарантировать правильность реализации. Пользователь может даже не подозревать о существовании модульных тестов. Приемочные тесты, напротив, используются разработчиками и пользователями, чтобы гарантировать правильную реализацию требований предметной логики. Пользователи должны понимать, как работают приемочные тесты, чтобы быть уверенными, что они действительно оценивают приемочные критерии.

Так же различаются уровни, проверяемые тестами. Как следует из названия, модульные тесты оценивают отдельные модули — элементы прило-

жения в изоляции друг от друга — например, отдельные классы. Как следствие, они не охватывают взаимодействий между модулями. Приемочные тесты оценивают более широкие части системы или даже систему целиком. Однако это лишь результат направленности тестов: цель приемочных тестов в том, чтобы обезопасить предметную логику — в конце концов, такое возможно, когда в тестировании участвуют не только отдельные классы. В этом отношении приемочные тесты также отличаются от интеграционных тестов. В центре внимания интеграционных тестов находятся только взаимодействия между компонентами, и они необязательно должны проверять критерии приемки программного обеспечения.

На техническом уровне модульные тесты являются практически «прозрачными ящиками». Прозрачными в том смысле, что являются простыми и понятными: целью этих тестов являются внутренние механизмы реализации и классы. Поэтому модульные тесты довольно «хрупкие»: если в ходе рефакторинга перенести некоторую функциональность в другие классы или изменить структуру классов, модульные тесты также должны быть адаптированы в соответствии с этими изменениями.

Приемочные тесты — это черные ящики: они проверяют правильность реализации с помощью программного или пользовательского интерфейса и не зависят от внутренних особенностей. Это обеспечивает им большую устойчивость: они должны сообщать об ошибках, только когда функция реализована неправильно. На этом уровне не важно, какие классы участвуют в работе и как эти классы выглядят.

В конце концов, главная ценность приемочных тестов в проверке того, что действительно ценно для пользователя, а именно — правильности реализации возможностей.

4.3.6. Окружения для тестирования

Внедрение методологии непрерывного развертывания упрощает создание окружений для тестирования благодаря автоматизации инфраструктуры. Однако остается одна проблема: необходимость присутствия в тестовых окружениях сторонних систем, не являющихся частью инфраструктуры. Эта проблема имеет несколько решений.

- Часто сторонние системы позволяют использовать тестовые системы, которые действуют подобно рабочим системам.

- Сторонние системы могут заменяться заглушками, имитирующими поведение рабочих систем.

В обоих случаях тестовое и рабочее окружения имеют различия. Это снижает надежность тестов — какие-то ошибки могут просто не определяться. Однако эта проблема не имеет исчерпывающего решения; рабочие и тестовые окружения никогда не будут абсолютно идентичны. Но методология непрерывного развертывания помогает в подобных сценариях еще по одной причине: поскольку процедура развертывания в рабочем окружении автоматизирована и риски минимизируются использованием дополнительных подходов (см. главу 7, «Развертывание — ввод в эксплуатацию»), риск, обусловленный ограниченной надежностью тестов, можно уменьшить. Если после исправления ошибка не определяется в процессе тестирования, его можно относительно быстро перенести в рабочее окружение.

4.4. Приемочные тесты через ГИП

Пользователи взаимодействуют с приложением посредством графического интерфейса (ГИП). Поэтому, тестируя приложение, они также будут взаимодействовать с графическим интерфейсом и наблюдать, насколько его поведение соответствует ожидаемому. Очевидно, что подобные тесты также выгодно было бы автоматизировать — это ведет нас к теме автоматизации тестов через ГИП, которая рассматривается в данном разделе. В таких тестах выполняется запись операций с ГИП и проверяется правильность отображаемых результатов.

4.4.1. Проблемы тестирования графического интерфейса

Однако это не самый правильный подход: в ходе тестирования теряется его семантика. В конце концов, функции графического интерфейса — лишь одна из особенностей приложения, и тесты должны проверять правильное функционирование этой особенности. Это различие становится особенно заметным при изменении графического интерфейса: тесты могут перестать выполняться из-за изменения внешнего вида интерфейса. Например, после изменений потребуется выбирать значения из списка, а не вводить их в текстовое поле. В таком случае тест потерпит неудачу, потому что не реализует выбор значений из списка. Однако суть приложения может остаться

ся прежней, поэтому в принципе тест должен был выполняться успешно. Соответственно, тесты через графический интерфейс очень уязвимы. Изменения в интерфейсе могут нарушить работу тестов даже в отсутствие ошибок в приложении. Это особенно верно, когда тестирующий записывает взаимодействия с графическим интерфейсом и позднее воспроизводит их в автоматическом режиме. Записанные взаимодействия опираются на конкретные элементы интерфейса. Когда они изменяются, появляется вероятность того, что тест потерпит неудачу.

4.4.2. Абстракции против хрупких тестов через графический интерфейс

Для преодоления этой проблемы в тесты графического интерфейса встраиваются абстракции, которые устраняют зависимость от конкретной реализации интерфейса. Если кнопка будет переименована или перемещена в другое место, достаточно изменить только слой абстракции — сам тест останется прежним. При такой организации подход к тестированию становится более похожим на процедурный подход, описываемый в разделе 4.6, в котором тесты описываются исключительно в текстовом виде. Однако когда такой подход становится слишком сложным, пользователи и инженеры-технологи перестают понимать тесты. Это противоречит цели проверки правильного функционирования приложения в ходе совместных приемочных испытаний разработчиками, пользователями и инженерами-технологами. Простые тесты графического интерфейса обычно не вызывают сложностей, потому что они лишь автоматизируют взаимодействия с интерфейсом. Дополнительные абстракции создают лишнюю завесу, что делает тесты более трудными для понимания.

Кроме того, процесс создания окружений для тестирования намного сложнее, потому что требуется развернуть приложение целиком, вместе с графическим интерфейсом, плюс установить базу данных, напоминающую рабочую. Более того, необходимо также воссоздать инфраструктуру, автоматизирующую тесты. В состав этой инфраструктуры, как правило, входит также клиентское программное обеспечение, соответствующие браузеры и инструменты автоматизации.

Несмотря на эти ограничения, тесты графического интерфейса могут стать ценными инструментами, потому что принцип их действия намного понятнее. Трудоемкие ручные операции должны быть автоматизированы, и такая автоматизация является основой методологии непрерывного раз-

вертывания. В любой момент пользователь, выполняющий такие тесты, сможет увидеть, что происходит в знакомом ему графическом интерфейсе.

4.4.3. Автоматизация с помощью Selenium

В качестве инструмента автоматизации в этой главе мы будем использовать Selenium [3]. Selenium позволяет запускать тесты удаленно, с помощью веб-браузера, имитирует управляющие воздействия пользователя и сравнивает отображаемые результаты с желаемыми.

4.4.4. Программный интерфейс веб-драйвера

Тесты могут быть реализованы в программном коде — этой цели служит специализированный программный интерфейс веб-драйвера. Программный интерфейс управляет удаленным веб-браузером. Поддерживаются практически все популярные браузеры, такие как Chrome, Safari, Internet Explorer и Firefox. Чтобы избежать необходимости выделять для каждого браузера отдельный компьютер, тесты выполняются в кластере, в этом случае на каждом компьютере установлен определенный набор браузеров. Управление кластером осуществляет сервер Selenium. В этом случае тесты могут выполняться с разными браузерами и позволяют выявлять различия в поведении разных браузеров. Кроме того, тесты могут выполняться параллельно, благодаря чему окончательные результаты становятся доступными быстрее. С помощью Selenium Grid тесты можно проводить сразу на нескольких компьютерах в сети. В этом случае не требуется, чтобы все браузеры были установлены на одном компьютере, а для особенно тяжелых тестов предоставляются дополнительные аппаратные средства, чтобы ускорить их выполнение.

4.4.5. Тестирование без веб-браузера: HtmlUnit

Selenium не требует обязательного использования веб-браузера. Как вариант, можно использовать HtmlUnit. Благодаря этой библиотеке на Java все необходимые взаимодействия с веб-приложением реализуются исключительно на языке Java и отпадает необходимость устанавливать браузер. В этом случае тесты выполняются на любом компьютере без установки дополнительного программного обеспечения. Поэтому в проекте примера используется именно этот подход.

4.4.6. Программный интерфейс веб-драйвера Selenium

Взаимодействие с веб-приложением — через браузер или HtmlUnit — осуществляется посредством программного интерфейса веб-драйвера (Web Driver API). Существуют официальные реализации этого интерфейса на языках программирования Java, Ruby, C#, Python и JavaScript, но, кроме того, сообщество развивает свои проекты поддержки многих других языков. Программист может реализовать тесты, используя этот программный интерфейс. Однако, как предполагается, приемочные тесты также должны быть понятны и, возможно, даже создаваться пользователями и тестировщиками.

4.4.7. Selenium IDE

Альтернативой программному интерфейсу является интегрированная среда Selenium IDE. Она предоставляет веб-интерфейс в браузере для записи взаимодействий с веб-приложением. Selenium IDE реализована как расширение для веб-браузера Firefox. Она позволяет вручную добавлять выражения к записанным взаимодействиям, чтобы исследовать правильность поведения приложения.

В идеале приемочные тесты представляют контракт между заказчиком и разработчиками. С помощью приемочных тестов заказчик выражает свои требования к функциональным возможностям. Поэтому необходимо, чтобы обе стороны понимали и могли выполнять приемочные тесты. Это делает Selenium IDE особенно привлекательной: с помощью инструмента заказчик сможет использовать приложение, определять и записывать желаемое поведение. Selenium IDE позволяет им делать это практически самостоятельно.

На рис. 4.3 изображен момент тестирования примера приложения в Selenium IDE. В данном случае имеется комплект, включающий несколько отдельных тестов. Первый тест называется `UserDoesNotExist` и проверяет отсутствие пользователя. С этой целью тест использует функцию поиска пользователя. Ожидаемый результат — пользователь отсутствует. Следующий тест выполняет регистрацию пользователя. Затем тест `RegisterUserAgain` выполняет попытку зарегистрировать пользователя во второй раз. Этот тест выбран на рис. 4.3, и подробную информацию о нем можно наблюдать справа в окне IDE. Первым действием тест

открывает домашнюю страницу с URL «/». Затем производится щелчок на ссылке Register User («Зарегистрировать пользователя»). В заключение вводятся данные о пользователе, и форма отправляется приложению. В ответ, например, приходит страница с сообщением already in use («уже зарегистрирован»), это мы видим в данном случае. Это сообщение об ошибке указывает, что регистрация не состоялась потому, что данный адрес электронной почты уже зарегистрирован.

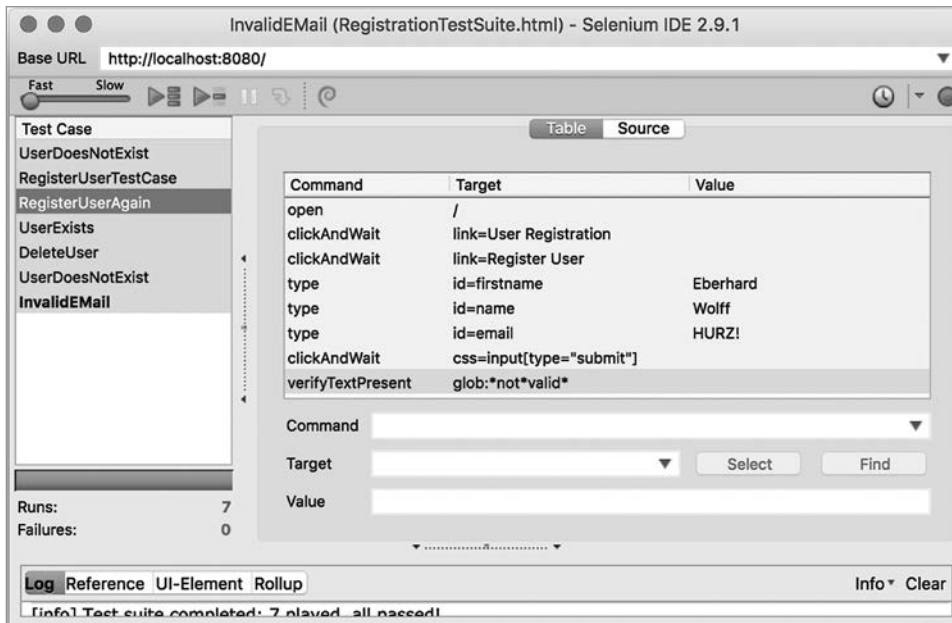


Рис. 4.3. Скриншот интегрированной среды Selenium IDE

Ниже перечислены остальные тесты.

- UserExists проверяет наличие зарегистрированного пользователя.
- DeleteUser удаляет пользователя.
- UserDoesNotExist проверяет отсутствие зарегистрированного пользователя.
- InvalidEmail проверяет отказ от регистрации в случае ввода недопустимого адреса электронной почты.

4.4.8. Проблемы автоматизации тестов через графический интерфейс

Однако, используя данный комплект тестов, мы получаем одну проблему: он состоит из нескольких шагов, зависящих друг от друга. Если пользователь не был успешно зарегистрирован на первом шаге, поиск и удаление учетной записи потерпят неудачу. Даже если ошибка возникнет на каком-то одном этапе, все остальные этапы также окончатся неудачей. Такие неопределенные ошибки затрудняют поиск причин, вызвавших проблему.

Очередной запуск комплекта тестов также станет причиной проблем: если в ходе предыдущего испытания был сохранен какой-то набор данных, они останутся на месте к моменту следующего запуска. Поэтому в конце тестирования эти данные нужно удалить. Однако если тест потерпел неудачу и завершился раньше времени, очистка не будет выполнена и система останется в состоянии, имевшем место на момент ошибки.

4.4.9. Выполнение тестов графического интерфейса

Selenium IDE может выполнять записанные тесты. В случае успеха такие тесты окрашиваются зеленым, а завершившиеся неудачей — красным цветом. Комплекты тестов и отдельные тесты хранятся в виде файлов HTML. Эти файлы содержат команды Selenium. Тесты можно запускать в автоматическом режиме, в процессе сборки. Для этого потребуется лишь установить сервер Selenium. Тестирование в этом случае запускается из командной строки, как показано ниже:

```
java -jar selenium-server-standalone-2.48.2.jar -html4Suite
*firefox http://localhost:8080/ RegistrationTestSuite.html
result.html.
```

Параметр `htmlSuite` устанавливает необходимый режим работы сервера. Далее следуют идентификатор браузера, базовый URL, имя файла с тестовыми данными и, наконец, имя файла с ожидаемыми результатами. Этот прием позволяет выполнять записанные тесты с помощью Selenium IDE в контексте сборки.

4.4.10. Преобразование тестов в программный код

Кроме всего прочего, тесты можно преобразовывать в программный код. Это дает возможность их оптимизировать, например, одноразовой реализа-

цией и многократным использованием функций в многочисленных тестах. Тесты также можно модифицировать, добавив защиту от изменений в графическом интерфейсе. Для выполнения таких тестов необходим фреймворк тестирования. В роли такого фреймворка используются Ruby с RSpec или Test::Unit, Python с unittest или C# с NUnit. Для Java можно использовать JUnit 4, JUnit 3 или TestNG. Несмотря на то что эти фреймворки изначально создавались для модульного тестирования, в данном случае они с успехом применяются для проведения приемочных испытаний. Для взаимодействий с веб-сайтом, где находится испытуемое приложение, тесты используют программный интерфейс веб-драйвера Selenium.

4.4.11. Изменение тестов вручную

Иногда возникает необходимость во вмешательстве разработчиков: в некоторых ситуациях сгенерированные тесты работают неправильно и требуют корректировки. Это касается, например, проверки корректности отображаемого результата. Кроме того, разработчик может унифицировать и упрощать код. Например, в примере приложения поиск клиента и настройка его данных выполняются несколько раз. В идеале подобные действия реализуются однократно и вызываются в нужных местах.

Модификации также повышают устойчивость тестов. Тесты должны подтвердить правильность получаемых результатов. Для этого они проверяют наличие определенных элементов — например, сообщения об ошибке. Однако если текст сообщения изменится, тест перестанет работать, даже при том что функционально все работает правильно. Этой проблемы можно избежать: например, если принять соглашение об использовании определенного класса CSS для отображения ошибки. Именно этот подход использован в сценарии на рис. 4.3 — выполняется поиск элемента `div` с CSS-классом `alert` и `alert-error`. Кроме того, проверяется соответствие возвращаемого сообщения об ошибке ожидаемому.

4.4.12. Тестовые данные

Кроме всего прочего, тесты должны генерировать тестовые данные. В проекте примера тестовые данные генерируются самим сценарием. Однако иногда это невозможно из-за слишком высокой сложности данных или потому, что на их создание требуется слишком много времени. В этом случае также может помочь разработчик, реализовав необходимые функции.

В конечном итоге будут созданы абстракции и вспомогательные данные, необходимые для реализации тестов. Так же как в случае с реализацией, проведение приемочных испытаний через графический интерфейс требует сотрудничества пользователей и разработчиков, потому что из-за изменений и реализации в программном коде заказчику сложно понять, что проверяется тестами. Именно в таком сотрудничестве состоит цель приемочных испытаний. Кроме того, способ, основанный на генерации программного кода, создает ряд проблем: поскольку сгенерированный код перерабатывается разработчиком, изменение тестов сопряжено с определенными трудностями. Нельзя просто сгенерировать код еще раз; сначала нужно изменить тест в Selenium IDE, а затем скорректировать сгенерированный код.

Тем не менее интегрированная среда Selenium IDE — важный инструмент, позволяющий выполнять HTML-сценарии. Ее удобно использовать для исследовательского тестирования (см. главу 6) и записи сценариев. С ее помощью тестировщики легко смогут воспроизводить тестовые ситуации и автоматизировать процедуры, хотя бы частично. Кроме того, описанный способ идеально подходит для реализации полностью автоматизированных тестов.

Еще одно преимущество данного подхода — тестирование интерфейса и выявление проблем с компоновкой элементов управления или несовместимостью браузеров.

4.4.13. Шаблон Page Object

Может так случиться, что тесты графического интерфейса перестанут работать из-за изменений в нем, например из-за изменения имен элементов интерфейса. С этой проблемой можно справиться — как часто случается в разработке программного обеспечения — путем добавления уровня абстракции. В данном конкретном случае доступ к элементам веб-страницы организуется с помощью шаблона проектирования Page Object («Объект страницы») [4]. Этот шаблон позволяет абстрагировать веб-интерфейс с точки зрения бизнес-подхода. Например, методы, такие как `registerCustomerWithEmailFirstNameName()`, могут внутренне представлять соответствующие визуальные элементы управления с данными и выполнять запросы. При изменении графического интерфейса достаточно скорректировать только эти методы вместо каждой строки кода.

4.5. Альтернативные инструменты тестирования графического интерфейса

Selenium широко и давно используется для реализации тестов графического интерфейса. Эта среда обладает множеством полезных особенностей и соответствует самым сложным требованиям. И все же существует несколько достойных альтернатив Selenium.

4.5.1. PhantomJS

В частности, в сообществе JavaScript популярен PhantomJS [5]. При использовании этого инструмента тесты пишутся на языке JavaScript. В остальном PhantomJS очень напоминает Selenium. Подобно Selenium, он также может использовать веб-драйвер для управления удаленным браузером.

4.5.2. Windmill

На первый взгляд Windmill [6] — очень интересное решение, позволяющее записывать тесты с помощью браузера. Кроме того, тестовые сценарии могут записываться на языке Python или JavaScript. Однако на данный момент проект Windmill прекратил развитие.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Для знакомства с особенностями приемочных испытаний на основе веб-интерфейса выполните следующие инструкции.

1. Установите и настройте Selenium IDE. Для этого посетите страницу <http://docs.seleniumhq.org/download/>.
2. Пример проекта можно найти по адресу: <http://github.com/ewolff/user-registration-V2>. Он понадобится для следующих шагов.
3. Установите Maven (см. <http://maven.apache.org/download.cgi>).
4. Выполните команду `mvn install` в каталоге `user-registration-V2`.

5. Выполните команду `mvn spring-boot:run` в подкаталоге `user-registration-application`.
6. Теперь попробуйте воспользоваться приложением по адресу <http://localhost:8080>.
7. Что случится, если попробовать дважды зарегистрировать один и тот же адрес электронной почты?
8. Что случится, если указать недопустимый адрес электронной почты?
9. На странице http://docs.seleniumhq.org/docs/02_selenium_ide.jsp#building-test-cases вы найдете информацию о том, как записывать и запускать тесты с помощью Selenium IDE.
10. В подкаталоге `user-registration/user-registration-acceptancetest-selenium/Selenium/en` найдите комплект тестов `RegistrationTestSuite.html`. Он включает проверку основных компонентов, участвующих в процедуре регистрации. Загрузите его в Selenium IDE и запустите.
11. Загрузите сервер Selenium Standalone Server; запустите комплект тестов с его помощью. Используйте для этого сценарий `runTest.sh`.
12. В Selenium IDE существует возможность генерировать программный код из комплектов тестов. Сделайте это и посмотрите, как выглядит этот код.
13. Необязательно: на основе этого кода реализуйте тот же комплект тестов на другом языке программирования по своему выбору.
14. В проекте `user-acceptancetests-selenium` вы найдете тесты на языке Java, созданные на основе кода, сгенерированного средой Selenium IDE.
15. Рассмотрите эти тесты.
16. Попробуйте выполнить их с помощью команды `mvn test`.
17. Сравните их с кодом, сгенерированным средой Selenium IDE.
18. Найдите унифицированные фрагменты.
19. Чем отличается сгенерированный и переработанный код, проверяющий результаты?
20. Перепишите тест, применив шаблон Page Objects.

4.6. Текстовые приемочные тесты

Как правило, требования записываются в виде обычного текста, соответственно, приемочные тесты также должны записываться как требования.

4.6.1. Behavior-Driven Development

В этом контексте прочно обосновался подход к разработке, основанной на функционировании (Behavior-Driven Development, BDD). В этом подходе письменное представление приемочных тестов формализовано до такой степени, что может выполняться автоматически. Методология BDD объединяет приемы разработки через тестирование (Test-Driven Development, TDD) с идеей *единого языка* (Ubiquitous Language) из методологии предметно-ориентированного проектирования (Domain-Driven Design, DDD) [7]. Методология разработки через тестирование предполагает создание тестов до фактической реализации, которую они должны тестировать. Конечно, в отсутствие соответствующей реализации тесты терпят неудачу. Когда тест начинает выполняться успешно, реализация проверяемой им особенности считается законченной.

Единый язык в DDD представляет общий язык для пользователей, разработчиков требований и программистов, фразы на котором можно найти в программном коде. Он включает основные термины предметной области. В BDD приемочные тесты записываются на языке, общем для всех: пользователей, тестировщиков и разработчиков.

Листинг 4.1. Пример истории в BDD¹

```
Narrative:
In order to use the website
As a customer
I want to register
So that I can login
```

Например, есть истории². Они описывают функциональные особенности в контексте предметной области. В листинге 4.1 представлена такая история для приложения регистрации. Она сообщает, что необходимо сделать

¹ Перевод истории:

История: Для использования веб-сайта в качестве клиента я должен зарегистрироваться, чтобы получить возможность войти. — *Примеч. пер.*

² Историями (narrative) в BDD называют тесты. — *Примеч. пер.*

(функцию) для обретения роли клиента. История определяет «in order to» (цель), «as a» (роль), «I want to» (функцию) и «so that» (результат). Однако такое описание позволяет «только» определять функциональные особенности, хотя и способствует лучшему пониманию приложения разработчиками и заказчиками. Его нельзя выполнить автоматически, как приемочный тест. Тем не менее это описание можно сделать частью теста JBehave.

Листинг 4.2. Приемочный тест BDD¹

Scenario: User registers successfully

Given a new user with email eberhard.wolff@gmail.com firstname Eberhard name Wolff

When the user registers

Then a user with email eberhard.wolff@gmail.com should exist

And no error should be reported

Давайте рассмотрим листинг 4.2. Здесь описывается конкретный сценарий. Сценарий принадлежит истории и определяет возможную последовательность событий в контексте истории. Он состоит из трех компонентов:

- «Given» (Дано) — описывает контекст сценария;
- «When» (Когда) — возникающее событие;
- «Then» (Тогда) — определяет ожидаемый результат.

Если имеется несколько компонентов одного типа, их можно объединить союзом «And» (И). Например, в листинге 4.2 определяется два ожидаемых результата: клиент с указанным адресом электронной почты должен существовать и никаких сообщений об ошибках не должно появиться.

Сценарий составлен на естественном языке; однако он соответствует формальной структуре, и его выполнение может быть автоматизировано. То есть ожидаемое поведение, описанное таким способом специ-

¹ Перевод сценария:

Сценарий: Успешная регистрация пользователя.

Дано: новый пользователь с адресом eberhard.wolff@gmail.com, именем Eberhard, фамилией Wolff.

Когда пользователь зарегистрирован, тогда пользователь с адресом eberhard.wolff@gmail.com должен существовать.

И никаких ошибок не должно появиться. — *Примеч. пер.*

алистом в предметной области, может быть проверено автоматическим тестом.

Кстати, такой тест вполне можно написать до фактической реализации тестируемого поведения. Инструмент тестирования только укажет, что код, необходимый для данного теста, еще не написан. То есть методика разработки через тестирование с успехом может применяться также в отношении приемочных тестов.

Листинг 4.3. Код для приемочного теста BDD

```
public class UserRegistrationSteps {
    private RegistrationService registrationService;
    private User customer;
    private boolean error = false;

    // Инициализация RegistrationService опущена

    @Given("a new user with email $email firstname $firstname"
        +" name $name")
    public void givenUser(String email, String firstname, String name) {
        user = new User(firstname, name, email);
    }

    @When("the user registers")
    public void registerUser() {
        try {
            registrationService.register(user);
        } catch (IllegalArgumentException ex) {
            error = true;
        }
    }

    @Then("a user with email $email should exist")
    public void exists(String email) {
        assertNotNull(registrationService.getByEMail(email));
    }

    @Then("no error should be reported")
    public void noError() {
        assertFalse(error);
    }
}
```

В листинге 4.3 демонстрируется код, необходимый для выполнения теста. Он использует фреймворк JBehave [8]. Текст внутри аннотаций `@Given`, `@When` и `@Then` указывает на соответствующий код, который должен быть

выполнен. В ходе выполнения теста из листинга 4.3 сначала вызовом метода `givenUser()` создается объект, представляющий пользователя. Затем в методе `registerUser()` производится регистрация пользователя. В заключение вызываются методы `exists()` и `noError()` для проверки ожидаемых результатов.

4.6.2. Адаптеры

Для реализации тестов в примере используется адаптер, который вызывает прикладную логику, соответствующую текстовому описанию приемочных тестов (рис. 4.4). Преимущество этого подхода в том, что он не требует запускать приложение целиком — достаточно простого приложения, поддерживающего тесты. Например, графический интерфейс не требуется. Соответственно, тесты выполняются очень быстро. Кроме того, тесты относительно легко запускаются на компьютере разработчика и даже внутри интегрированной среды разработки. Используя адаптер, разработчики могут реализовать только функции, необходимые для тестов. То есть реализация тестов упрощается (см. рис. 4.4).

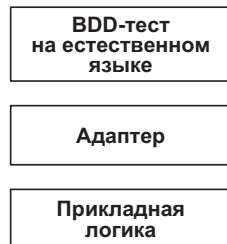


Рис. 4.4. Приемочные тесты в текстовом виде с адаптером для доступа к прикладной логике

На рис. 4.5 изображена альтернативная схема: в этом случае приемочные тесты работают через адаптер с веб-интерфейсом. Тесты все еще могут записываться на естественном языке, но действуют через графический веб-интерфейс. Пример описания такого теста приводится в листинге 4.4. Как можно заметить, в данном случае тестирование осуществляется на уровне графического веб-интерфейса, поэтому также обнаруживает ошибки интерфейса. Однако эти тесты определяются на более высоком уровне абстракции, чем тесты в Selenium (см. рис. 4.5).

Листинг 4.4. Пример текстового описания приемочного теста, выполняемого через графический интерфейс¹

```
Given user is on the homepage
When the user enters eberhard.wolff@gmail.com as email
And submits the search form
Then 4 user should be found
```

Главное отличие — этот тест не ссылается на элементы интерфейса по конкретным именам и не определяет внешний вид веб-сайта, если искомый пользователь не найден. Эта логика реализуется на уровне адаптера. Это упрощает реализацию теста и делает его более устойчивым в сравнении с тестами Selenium. Изменения в графическом интерфейсе можно компенсировать корректировкой кода адаптера, чтобы тест не потерял работоспособности.

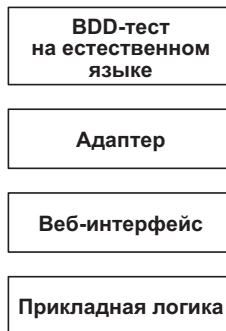


Рис. 4.5. Приемочные тесты в текстовом виде с адаптером, действующим через графический веб-интерфейс

Кроме того, тесты одновременно проверяют работу графического веб-интерфейса, тогда как другие приемочные тесты в текстовом виде взаимодействуют с прикладной логикой напрямую. Однако это также означает, что должно быть доступно соответствующее окружение с действующим веб-сервером.

¹ Перевод текстового описания:

Дано: пользователь находится на домашней странице.

Когда пользователь введет адрес электронной почты eberhard.wolff@gmail.com и отправит форму поиска, тогда 4 пользователя должны быть найдены. — *Примеч. пер.*

4.7. Альтернативные фреймворки

Кроме JBehave существует еще ряд фреймворков, которые можно использовать для выполнения тестов в виде текстового описания.

- Cucumber [9] поддерживает разные языки программирования. Тесты записываются на естественном языке, как в случае с JBehave.
- RSpec [10] предлагает поддержку аналогичного подхода для Ruby. Однако при его использовании тесты записываются не на естественном языке, а на языке Ruby.
- Jasmine [11] — фреймворк для JavaScript, позволяющий записывать требования на языке JavaScript. Код легко читается, но для лиц, не связанных с разработкой, он может оказаться не таким непонятным, как тесты для фреймворков, поддерживающих описание на естественном языке.

При использовании фреймворка JGiven [12] тесты можно записывать на языке Java, но результаты выводятся в виде простого текста. То есть программист относительно легко сможет написать тесты, а специалист в предметной области — понять их суть. Однако в данном случае специалист в предметной области едва ли сможет сам писать тесты.

Инструменты, такие как RSpec или Jasmine, заявляют о поддержке BDD, но не поддерживают естественный язык. Вместо этого они предлагают писать тесты на языке программирования. Недостаток такого подхода в том, что заказчик не сможет самостоятельно разобраться в программном коде. То есть первоначальная цель приемочных испытаний — совместное определение требований и работа над автоматизацией их проверки — остается недостигнутой.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Проект примера можно найти по адресу: <http://github.com/ewolff/user-registration-V2>. Загрузите его и выполните следующие инструкции.

1. Установите Maven (см. <http://maven.apache.org/download.cgi>).
2. Затем выполните команду `mvn install` в каталоге `user-registration-V2`.

После этого вы сможете выполнить приемочные тесты, записанные в текстовом виде.

- ▲ Выполните тест в подкаталоге `user-registration-acceptancetest-jbehave` командой `mvn integration-test`.
- ▲ Попробуйте нарушить работу теста, внося в него изменения. Код теста можно найти в файле `src/main/resources/com/ewolff/user-registration/user_registration_scenarios.story`.
- ▲ Откройте страницу <http://jbehave.org/reference/web/stable/using-selenium.html> и попробуйте реализовать тест из листинга 4.4, используя веб-драйвер. В качестве примера используйте существующий тест.
- ▲ Прочитайте вводную статью с описанием Cucumber [13], ее можно найти по адресу: <https://github.com/cucumber/cucumber/wiki/tutorials-and-related-blog-posts>. Так как Cucumber поддерживает много языков программирования, с его помощью можно тестировать приложения на Java, а также приложения на многих других языках.

4.8. Стратегии приемочных испытаний

Чтобы извлечь выгоду из приемочных испытаний, важно использовать правильный инструмент. Подходы, представленные в этой главе, основаны на тестировании через графический интерфейс или на использовании текстового описания приемочных критериев. Технические аспекты подходов не так важны: главная цель приемочных испытаний — наладить общение между пользователями, инженерами-технологами и разработчиками. Если пользователи или инженеры-технологи не смогут пользоваться инструментом, они отложат его, и применение соответствующего подхода окажется безуспешным. Поэтому, выбирая инструмент для организации приемочных испытаний, очень важно выбрать технологию, простую и понятную для пользователей и тестировщиков. В конце концов приемка программного обеспечения заказчиком выполняется на основе приемочных испытаний.

4.8.1. Выбор инструмента

Некоторые заказчики и инженеры-технологи оформляют протоколы тестирования в Excel. Эти протоколы содержат описания отдельных этапов тестирования. Аналогично, существуют проекты, где Excel используется для определения желаемых результатов, соответствующих исходным

данным. В такой ситуации имеет смысл использовать файлы Excel в роли входных данных приемочных испытаний. Это позволит использовать одни и те же инструменты. Просто потребуются наладить автоматический анализ данных в электронных таблицах Excel, которые должны оформляться в соответствии с набором некоторых формальных правил. Это гораздо проще, чем обучать пользователей правилам и приемам работы с новым инструментом.

Техническая реализация этого приема не слишком трудоемка: существует, например, фреймворк Framework for Integrated Tests (Fit) [14], который следует именно этому подходу. Он доступен для разных платформ. Fit принимает файлы HTML с определениями тестов. А большинство инструментов из состава Microsoft Office, таких как Excel и Word, поддерживают экспорт документов в формат HTML. Они могут использоваться как приемочные тесты для таких инструментов.

Альтернативой может служить concordion [15]. Этот инструмент также использует файлы HTML, но код разметки в таких файлах требуется дорабатывать. Результат экспорта из Excel или Word нельзя использовать непосредственно.

Все эти инструменты значительно упрощают общение с заказчиком по поводу приемочных испытаний. И все же иногда невозможно организовать совместную работу заказчика и разработчиков над приемочными тестами. Но даже в таких случаях имеет смысл создавать автоматизированные приемочные тесты, потому что они гарантируют правильную работу прикладной логики. Совместная работа разработчиков и заказчиков над приемочными тестами упрощает их создание и дает возможность согласовать круг функциональных возможностей. Когда такое сотрудничество невозможно, работа осложняется, но не становится невыполнимой. Разработчики могут записать требования заказчика в виде тестов. Наконец, реализация действующего кода также осуществляется на основе требований заказчика.

4.8.2. Быстрая обратная связь

Одной из важнейших целей методологии непрерывного развертывания является быстрое получение обратной связи. Когда разработчик сохраняет код, содержащий ошибку, конвейер непрерывного развертывания должен обнаружить ее максимально быстро. Поэтому, проводя приемочные испытания, предпочтительнее сначала проверять простейшие случаи использования всех функциональных особенностей и только потом перехо-

доть к более глубокому тестированию. Если функция просто не работает, эта проблема будет обнаружена в самом начале. В этом случае детализированные тесты все равно потерпели бы неудачу, поэтому вполне можно обойтись без их выполнения. Такой подход рекомендуется использовать на разных этапах конвейера непрерывного развертывания.

4.8.3. Охват тестами

Цель приемочных тестов состоит не в том, чтобы максимально полно охватить ими программный код. Идеальные приемочные тесты структурированы так, что их успешное выполнение равносильно приемке программного обеспечения заказчиком. Поэтому основное внимание должно уделяться не крайним случаям, а функциям, особенно важным для заказчика.

Важно также рассматривать приемочные испытания в комплексе с исследовательским тестированием (глава 6). Автоматизированные приемочные тесты освобождают тестировщиков от рутинных задач, позволяя им сосредоточиться на исследовательском тестировании. Поэтому полная автоматизация всех тестов не является обязательной — ручное тестирование все равно будет присутствовать в том или ином виде.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Загляните в свой текущий проект.

- ▲ Какие тесты в нем имеются?
- ▲ В каком виде они записаны? В виде инструкции по тестированию в файле Excel? Как далеки они от полной автоматизации?
- ▲ Какой инструмент (Selenium, JBehave, concordion, Fit) лучше всего подойдет для их автоматизации? Какую долю тестов можно автоматизировать с его помощью?

4.9. В заключение

Для реализации приемочных тестов обычно используется тестирование через графический интерфейс. На первый взгляд такая стратегия выглядит безупречной: в конце концов, когда тестирование выполняется

вручную, также используется графический интерфейс, поэтому кажется, что нет ничего лучше, чем автоматизировать эти взаимодействия. Интегрированная среда Selenium IDE позволяет записывать и автоматизировать последовательности взаимодействий с интерфейсом. Однако все не так просто: тесты, использующие графический интерфейс, оказываются очень хрупкими — их работоспособность легко могут нарушить даже самые невинные изменения в графическом интерфейсе. Преобразование тестов графического интерфейса в код дает возможность их оптимизации, однако в этом случае записанные взаимодействия и тесты существенно различаются, из-за чего пользователь может не понять, охватывают ли тесты соответствующую прикладную логику. Кроме того, данный подход не позволяет определить приемочные критерии до реализации фактического кода. Наконец, автоматизация ручных тестов может гарантировать их выполнение. Однако ошибка в одном тесте зачастую приводит к неудаче всех последующих. Ошибка, возникшая на этапе подготовки тестовых данных, может, например, вызвать массу последующих ошибок из-за недоступности данных. Помимо этого, подготовка больших объемов тестовых данных затруднена при использовании описываемого подхода.

Приемочные тесты, записываемые в простом текстовом виде, лишены этих недостатков, так как используют формулировки на естественном языке. Эти тесты не требуют наличия функционирующего кода. Тест можно написать до фактической реализации, проверяемой им. Каждый тест обычно охватывает какой-то один сценарий — если какая-то ошибка просочилась в прикладную логику, неудачу потерпит только один сценарий.

Неплохим компромиссом является использование тестов через графический интерфейс, записываемых в простом текстовом виде. В конечном итоге главной целью приемочных испытаний является налаживание общения между заказчиками и разработчиками. Если заказчик доверяет только тестам графического интерфейса, которые он самостоятельно сможет прочитать, понять и выполнить, — это важный аргумент в пользу данного подхода. Если приемочные критерии записываются в виде таблиц Excel, хорошей альтернативой может оказаться concordion или Fit, потому что, как известно, клиент всегда прав...

Однако, даже если полноценное общение с заказчиком по поводу реализации приемочных тестов невозможно, автоматизация приемочного тестирования не теряет своей важности, потому что помогает гарантировать правильность выполнения прикладной логики.

Ссылки

1. <http://jbehave.org/>
2. <https://martinfowler.com/bliki/TestPyramid.html>
3. <http://docs.seleniumhq.org/>
4. <https://martinfowler.com/bliki/PageObject.html>
5. <http://phantomjs.org/>
6. <https://github.com/windmill>
7. Eric Evans: *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003, ISBN 978-0-32112-521-7¹.
8. <http://jbehave.org/>
9. <https://cucumber.io/>
10. <http://rspec.info/>
11. <https://jasmine.github.io/>
12. <http://jgiven.org/>
13. <https://cucumber.io/>
14. <http://fit.c2.com/>
15. <http://concordion.org/>

¹ Эрик Эванс. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. М.: Вильямс, 2016. ISBN: 978-5-8459-1942-7. — Примеч. пер.

5

Тестирование пропускной способности

5.1. Введение

Тестирование пропускной способности позволяет гарантировать, что приложение обладает достаточной производительностью для обслуживания определенного количества пользователей. В разделе 5.2 обсуждаются мотивы для тестирования пропускной способности и основные проблемы, с которыми приходится сталкиваться при реализации этих тестов. В разделе 5.3 рассматриваются конкретные реализации тестов пропускной способности. Gatling — инструмент, специально созданный для тестирования пропускной способности, — обсуждается в разделе 5.4, где также демонстрируется применение этого инструмента для реализации тестирования пропускной способности примера приложения. Поскольку ни один инструмент не предлагает оптимального решения на все случаи жизни, в разделе 5.5 будут представлены некоторые дополнительные альтернативы.

5.1.1. Тестирование пропускной способности: пример

Важнейшей особенностью тестов пропускной способности, описываемых в этой главе, является их автоматизация. До введения автоматизированного тестирования пропускной способности в компании «Big Money Online Commerce Inc.», представленной в разделе П.2, наблюдались проблемы с некоторыми версиями, выражающиеся в неожиданном снижении производительности. Благодаря автоматизации тестирования пропускной способности теперь каждое изменение проходит все необходимые испытания. Проблемы с производительностью можно увязать с конкретными изменениями в коде, а также с конкретными случаями использования. Прежде какие-то выводы о проблемах с производительностью можно было сделать только на основе ручного тестирования, этот вид тестирования проводился

нерегулярно и не являлся частью стандартного комплекса тестов. Поэтому было очень сложно получить исчерпывающую информацию о производительности.

5.2. Тестирование пропускной способности — как?

Прежде всего важно пояснить некоторые термины.

○ Производительность.

Скорость обработки системой определенного запроса. При плохой производительности пользователь вынужден будет долго ждать реакции системы на свой запрос.

○ Пропускная способность.

Количество запросов, обрабатываемых системой в единицу времени. Чем ниже пропускная способность, тем меньше пользователей смогут пользоваться системой одновременно.

Главная проблема в этой области — нелинейность. Например, в линейной системе, если 200 пользователей нагружают сервер на 25%, тогда 400 пользователей должны нагрузить его на 50%. Однако, поскольку системы не являются линейными, результат будет выглядеть иначе. Это существенно ограничивает информационную ценность тестов пропускной способности: тесты выполняются в окружении, отличающемся от рабочего, и используют данные, не соответствующие рабочим данным. Поэтому отражают реальную нагрузку лишь отчасти.

При наличии нелинейного эффекта во время эксплуатации могут возникать проблемы даже при успешном выполнении тестов пропускной способности. Например, возможно, что в базе не используются индексы. При работе с небольшими наборами тестовых данных это не вызывает проблем, но может стать причиной существенного падения производительности при работе с данными, накопленными в рабочем окружении.

5.2.1. Цели тестирования пропускной способности

Тесты пропускной способности служат до определенной степени гарантией высокой производительности и пропускной способности приложений. Производительность и пропускная способность — важные нефункцио-

нальные требования: если приложение действует недостаточно быстро, это приемлемо только до определенной степени. С некоторого момента низкая производительность оказывается недопустимой.

5.2.2. Данные и окружения

Иногда желательно выполнять тестирование пропускной способности с полным объемом данных из рабочего окружения и в окружении, близком по своим характеристикам к рабочему. Однако зачастую это невозможно из-за отсутствия необходимого аппаратного обеспечения.

Кроме того, данные должны соответствовать реальным рабочим данным — иначе приложение будет вести себя нереалистично. В этом случае следует использовать специальный генератор тестовых данных. Это позволит относительно просто генерировать большие объемы разных тестовых данных.

5.2.3. Тестирование производительности должно выполняться только в конце реализации?

Достижение необходимой производительности является фундаментальной проблемой: достоверно оценить производительность приложения можно, если имеются все прикладные функции и все они реализованы правильно. Только тогда приложение делает то, что от него требуется. Пока реализация неполная, отсутствие каких-то частей может приводить к результатам тестирования, далеким от реальности. Однако когда тестирование наконец начинается, но приложение не демонстрирует обещанной производительности, зачастую уже слишком поздно вносить коренные изменения, а кроме того, в ходе оптимизации в приложение могут просочиться логические ошибки. Для поиска таких ошибок снова требуется выполнить приемочные тесты, а затем повторить весь комплекс тестов пропускной способности, чтобы оценить улучшение производительности.

5.2.4. Тесты пропускной способности = управление рисками

По сути, тесты пропускной способности представляют форму управления рисками. Наша задача — убедиться в достижении пропускной способности, необходимой для комфортной работы пользователя. Фундаментальной

основой для этого является построение модели: модель должна отражать истинное поведение приложения в рабочем окружении и поведение пользователей. Достичь этого не так просто, потому что приложение может использоваться по-разному. Когда приложение, похожее на существующие, уже находится в рабочем окружении, параметры его функционирования можно интегрировать в тесты. Однако если реализуется совершенно новое приложение, тесты пропускной способности могут основываться только на прогнозах и догадках.

5.2.5. Имитация поведения пользователей

Для имитации поведения пользователей важно учитывать не только их количество, но также типичный порядок работы с приложением и то, какие функциональные особенности они используют. В зависимости от приложения мелкие различия могут оказывать существенное влияние на производительность и, соответственно, на выполнение тестов пропускной способности.

Зачастую имеет смысл исследовать производительность в разных сценариях использования: для веб-сайта электронной коммерции такими сценариями могут быть, например, поиск в каталоге товаров, выбор разных товаров и их покупка. Это поможет конкретизировать требования к производительности.

- Как долго может осуществляться поиск?
- Как много времени может быть потрачено на совершение покупки? Возможно в этом сценарии допустимо потратить больше времени, чем на начальный поиск.
- Как много пользователей может одновременно пользоваться веб-сайтом?

5.2.6. Документирование требований к производительности

Тестирование производительности возможно только при наличии конкретных и измеримых требований в отдельных сценариях. Такие требования, как «быстро», «примерно 5 миллисекунд» или «без заметной задержки», следует заменить более конкретными, такими как «меньше 5 миллисе-

кунд». Опираясь на подобные значения, можно с помощью различных инструментов реализовать требуемую модель сценария.

Также важно, чтобы требования были реалистичными. Если предъявить слишком строгие требования, потребуется чрезмерная оптимизация приложения. Это не только увеличит затраты на реализацию, но даже потребует покупки дорогостоящего оборудования. Если, наоборот, требования окажутся слишком мягкими, программное обеспечение может оказаться непригодным к использованию. То есть должен соблюдаться определенный баланс. В крайних случаях может потребоваться и оказаться вполне приемлемым более длительное время обработки.

5.2.7. Аппаратное обеспечение для тестирования пропускной способности

При создании модели важно определить требования к аппаратному обеспечению для тестирования: если рабочее окружение недоступно, следует использовать другое окружение, которое как минимум позволит делать выводы об ожидаемой производительности в рабочем окружении. Например, если приложение предполагается запускать в кластере, вполне возможно, что для тестирования пропускной способности будет достаточно одного узла кластера. Кроме того, количество имитируемых пользователей должно выбираться с учетом количества пользователей, приходящихся на один узел кластера в рабочем окружении. Также в ходе тестирования необходимо проверить возможность масштабирования приложения по горизонтали — то есть с увеличением количества узлов нагрузка между ними должна распределяться равномерно. Однако нелинейные эффекты могут сделать весь подход бесполезным.

Для получения надежных результатов оборудование для тестирования пропускной способности по своим параметрам должно быть как можно ближе к оборудованию, действующему в рабочем окружении. Это относится не только к серверному оборудованию, которое позволит судить о производительности в рабочем окружении, но также к оборудованию клиентов и организации сети.

Клиентская часть должна выполняться в отдельной системе. В противном случае сервер и генератор нагрузки будут вынуждены конкурировать друг с другом за ресурсы системы. Это не позволит получить достоверную картину, потому что условия во время тестирования будут существенно различаться.

Сетевая инфраструктура должна как можно точнее повторять сетевую инфраструктуру в рабочем окружении: если в рабочем окружении сервер и клиент разделены разными маршрутизаторами, коммутаторами и брандмауэрами, аналогичные средства должны участвовать в тестировании пропускной способности. Если инфраструктуру невозможно воспроизвести в точности, необходимо хотя бы симитировать задержки в сетевых взаимодействиях с помощью симуляторов.

Для имитации значительной нагрузки желательно установить генераторы нагрузки на несколько компьютеров в сети и координировать их работу. Это поможет воспроизвести нагрузку, не ограниченную мощностью одного компьютера.

5.2.8. Облачные решения и виртуализация

В случае с тестами пропускной способности автоматизация непрерывного развертывания приобретает особую ценность: благодаря ей сокращается время на подготовку окружения для тестирования пропускной способности. Также в качестве инфраструктуры можно использовать облачное решение. Облачное окружение можно арендовать только на время, необходимое для тестирования. Это уменьшает затраты на создание тестового окружения, сходного с рабочим окружением, а также позволяет получить окружение, по своим характеристикам близкое к рабочему, ничего не приобретая, а только арендуя вычислительные мощности. В облачном окружении также упрощается проверка влияния изменения мощности серверов на работу приложения: облачный «компьютер» можно перезапустить заново и увеличить количество процессоров в нем или объем оперативной памяти.

Технологии виртуализации позволяют получить нечто подобное в вашем вычислительном центре. Однако в этом случае должны иметься ресурсы, достаточные для создания окружения, подобного рабочему, — то есть потребуется приобрести дополнительные компьютеры, тогда как в общедоступном облаке их можно просто арендовать на ограниченный срок.

Конечно, производительность виртуального окружения несопоставима с производительностью физического аппаратного окружения. Кроме того, облачная инфраструктура используется многими клиентами, чья активность может влиять на результаты тестирования. Однако в наши дни многие рабочие окружения также часто размещаются в виртуальной среде и, соответственно, подвержены тем же эффектам.

5.2.9. Минимизация рисков за счет непрерывного тестирования

Как уже отмечалось выше, главная проблема тестирования пропускной способности — слишком маленький отрезок времени в конце разработки проекта, когда все еще можно внести какие-то изменения. Однако по-настоящему проверить пропускную способность приложения можно только в конце проекта, когда будут реализованы все функциональные возможности, так как только в этом случае можно надежно оценить его производительность.

Разумеется, подход, когда тестирование пропускной способности производится только в конце, плохо укладывается в методологию непрерывного развертывания. Запуск тестов не в конце разработки, а после каждого изменения позволяет гарантировать высокую производительность в любой момент. Вместе с новыми функциями могут добавляться новые тесты пропускной способности, проверяющие производительность новых возможностей. Благодаря этому можно относительно быстро выяснить, насколько пострадала производительность приложения, и приступить к необходимой оптимизации, сосредоточив внимание только на последних изменениях, которые, скорее всего, являются причиной проблем, выявленных в ходе тестирования пропускной способности, так как до этих изменений тесты выполнялись успешно. Этот подход также гарантирует достижение требуемой производительности всего приложения, которую в противном случае можно было бы узнать только на заключительном этапе тестирования. Это существенно упрощает управление рисками, связанными с производительностью.

5.2.10. Тестирование пропускной способности — целесообразно ли?

Основная проблема тестирования пропускной способности заключается в правильном моделировании производительности и поведения рабочего окружения. Однако никакая модель не может точно воспроизвести все характеристики, касающиеся аппаратного обеспечения и тестовых данных. У реальных пользователей всегда возникают порой самые неожиданные идеи, к тому же на тестовом окружении приходится экономить. Поэтому тесты пропускной способности всегда должны увязываться с результатами мониторинга рабочего окружения. Это позволит своевременно приводить тесты в соответствие с поведением рабочего окружения и оценивать

его производительность. Полученные результаты можно использовать для определения стратегии оптимизации системы.

Конвейер непрерывного развертывания позволяет быстрее и надежнее переносить изменения в рабочее окружение. При этом уменьшается риск появления проблем с производительностью, потому что разработчики получают возможность реагировать намного быстрее. Когда производительность рабочего окружения находится под постоянным контролем и разработчики способны достаточно быстро реагировать на возникающие проблемы, отпадает необходимость затрачивать большие усилия на тестирование пропускной способности. Риск снижается еще больше благодаря поступлению изменений небольшими порциями, которые легко откатить. То есть, когда в новой версии, развернутой в рабочем окружении, обнаруживается проблема, связанная с производительностью, можно использовать меры, описанные в главе 7, «Развертывание – ввод в эксплуатацию», для выяснения эффектов, возникших во время развертывания, и возврата к предыдущей версии программы. Это также подчеркивает ограниченную прогностическую ценность тестов пропускной способности из-за нелинейности эффектов.

5.3. Реализация тестов пропускной способности

Для реализации тестов пропускной способности применяются подходы, похожие на использовавшиеся в приемочных испытаниях.

- Тесты можно реализовать через отдельный программный интерфейс в адаптере, экспортирующий функциональность приложения конкретно для нужд тестирования пропускной способности (рис. 5.1). Нагрузка создается генератором нагрузки, который вызывает программный интерфейс. Это позволяет реализовать функциональные возможности специально для тестирования пропускной способности – например, инициализацию тестовых данных. Недостатком такого подхода является несоответствие API фактическим сценариям использования. Поэтому измеренная производительность не будет в точности совпадать с производительностью в рабочем окружении, что является еще одним фактором, снижающим информационную ценность тестов пропускной способности. Кроме того, некоторые разделы приложения, такие как графический интерфейс, вообще выпадают из тестирования.

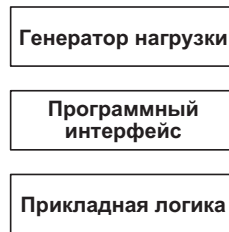


Рис. 5.1. Тестирование пропускной способности через программный интерфейс

- Альтернативное решение заключается в тестировании через интерфейс приложения, используемый в других случаях. Это может быть веб-интерфейс, графический интерфейс пользователя или прикладной интерфейс веб-служб. Генератор нагрузки использует этот интерфейс и таким способом создает желаемую нагрузку (рис. 5.2). То есть приложение используется в точности как в рабочем окружении. Однако это усложняет реализацию тестов пропускной способности, потому что приходится использовать интерфейс, не предназначенный для тестирования. Тест может даже действовать внутри браузера, измеряя и отображая нагрузку с помощью кода на JavaScript.

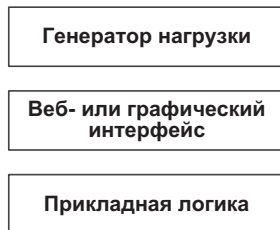


Рис. 5.2. Тестирование пропускной способности через графический веб-интерфейс

Тесты пропускной способности должны давать однозначный результат. Приложение либо соответствует, либо не соответствует требованиям. Если требования не выполняются, тест должен терпеть неудачу. Это гарантирует, что потенциальная проблема будет замечена и исправлена. Когда просто отображается отчет с результатами, разработчик должен внимательно проанализировать его, чтобы определить, соответствуют ли измеренные показатели предъявляемым требованиям. В какой-то момент

кто-то обязательно забудет выполнить этот шаг, и тогда вполне возможно, что проблема производительности не будет выявлена вовремя. Соответственно, существует вероятность, что результаты тестирования останутся без внимания.

Тем не менее всегда хорошо иметь отчеты, потому что подробный отчет может пригодиться для определения причин, вызывающих проблемы с производительностью.

Эта тема подробно будет обсуждаться в главе 8, «Эксплуатация».

5.4. Тестирование пропускной способности с помощью Gatling

Для демонстрации тестирования пропускной способности мы использовали в примере приложения инструмент Gatling [1]. Этот инструмент служит генератором нагрузки, имитирующим действия пользователей, и, соответственно, прекрасно подходит для тестирования пропускной способности. Gatling обладает следующими достоинствами.

- Реализован на языке программирования Scala. То есть может выполняться под управлением виртуальной машины Java (Java Virtual Machine, JVM) и в то же время пользоваться поддержкой параллельного программирования, которой обладает Scala. В частности, для генератора нагрузки важно уметь имитировать действия сразу нескольких пользователей, чтобы смоделировать более или менее реалистичную нагрузку.
- Тесты записываются на выразительном предметно-ориентированном языке. Это упрощает создание собственных сценариев и расширение существующих, в которых можно использовать всю широту возможностей языка программирования Scala.

А теперь вернемся к примеру приложения регистрации клиентов. Первым делом необходимо настроить регистратор Gatling (рис. 5.3). Он играет роль прокси-сервера для браузера и записывает все запросы, посылаемые им. Основными параметрами настройки являются номер порта регистратора и место для сохранения записанного сценария.

На рис. 5.4 изображен пример с записанными данными. Здесь регистратор зафиксировал обращение к начальной странице примера приложения и процесс регистрации нового пользователя. На данном этапе данные для

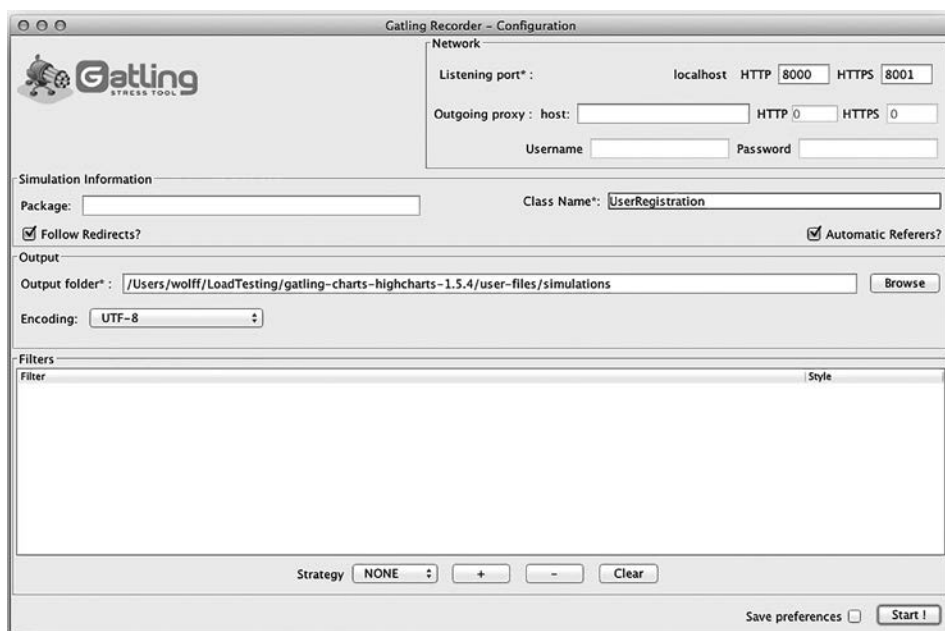


Рис. 5.3. Настройки регистратора Gatling Recorder

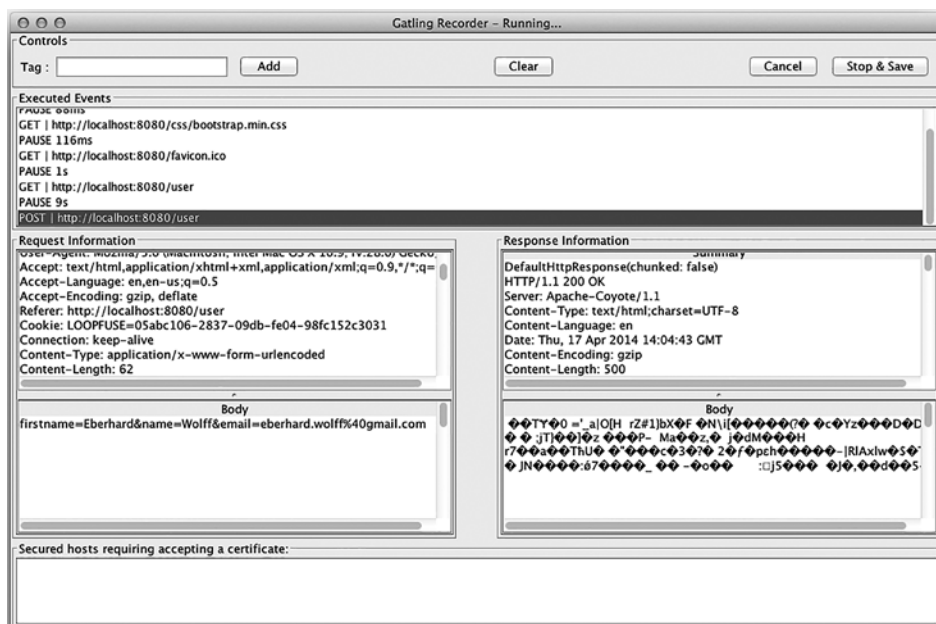


Рис. 5.4. Пример данных, записанных регистратором Gatling Recorder

регистрации передаются в параметрах запроса. Ответ сервера является страницей HTML. Однако она упакована и поэтому не читаема. Регистратор записывает взаимодействия в виде исходного кода на Scala.

В листинге 5.1 приводится полученный код после некоторой доработки вручную. В сценарий был добавлен `emailFeeder` — объект, позволяющий генерировать новый адрес электронной почты при каждом следующем запуске теста. В данном случае уникальность достигается за счет использования UUID, то есть строки с глобально-уникальным идентификатором, состоящим из цифр. Несмотря на то что во время записи сценария использовался фиксированный адрес, этот тест будет генерировать новый адрес при каждом следующем запуске. То есть `emailFeeder` внедряет соответствующие тестовые данные. Если бы все тесты выполнялись с одним и тем же адресом, это привело бы к проблеме, потому что в приложении не может существовать двух пользователей с одинаковыми адресами электронной почты. Когда два пользователя одновременно попытаются зарегистрироваться под одним и тем же адресом, возникнет ошибка. В ходе тестирования пропускной способности имитируется одновременная работа сразу нескольких пользователей, соответственно, для всех имитируемых пользователей должны выбираться разные адреса электронной почты. Подобная параметризация тестовых данных часто бывает необходима для имитации действий большого количества пользователей.

Листинг 5.1. Код теста пропускной способности на Scala DSL, созданный инструментом Gatling

```
class UserRegistration extends Simulation {
  val emailFeeder = new Feeder[String] {
    override def hasNext = true
    override def next: Map[String, String] = {
      val email =
        scala.math.abs(java.util.UUID.randomUUID.
          getMostSignificantBits)
        + "_gatling@dontsend.com"
      Map("email" -> email)
    }
  }
  val httpProtocol = http
    .baseUrl("http://127.0.0.1:8080")
    .acceptHeader(
      "text/html,application/xhtml+xml," +
      "application/xml;q=0.9,*/*;q=0.8")
    .acceptEncodingHeader("gzip, deflate")
    .acceptLanguageHeader("en,en-us;q=0.5")
}
```

```

        .connection("keep-alive")
        .header("Cache-Control", "max-age=0")
val formHeader = Map(
    "Content-Type" -> "application/x-www-form-urlencoded")
val scn = scenario("Registration")
    .repeat(10) {
        (
            exec(http("GET index")
                .get("/")
                .pause(88 milliseconds)
                .exec(http("GET css")
                    .get("/css/bootstrap.min.css"))
                .pause(1)
                .exec(http("GET form")
                    .get("/user"))
                .pause(7)
                .feed(emailFeeder)
                .exec(http("POST user data")
                    .post("/user")
                    .headers(formHeader)
                    .formParam("firstname", "Eberhard")
                    .formParam("name", "Wolff")
                    .formParam("email", "${email}"))
                .pause(4)
                .exec(http("POST delete user")
                    .post("/userdelete")
                    .headers(formHeader)
                    .formParam("email", "${email}")))
        )
    }
setUp(scn.inject(rampUsers(5) over (10 seconds))).
protocols(httpProtocol)
}

```

Далее выполняется настройка объекта `httpProtocol`, представляющего параметры протокола HTTP, такие как разные заголовки и URL-приложения. Переменная `formHeader` содержит дополнительные заголовки, необходимые для передачи данных формы. Фактический сценарий достаточно прост: тест выполняется десять раз. Сначала он открывает домашнюю страницу приложения ("GET index") и читает используемые стили CSS ("GET css"). Затем запрашивает форму ("GET form") и отправляет данные ("POST user data"). В заключение только что созданная учетная запись удаляется ("POST delete user"). Между этими действиями выполняются паузы, имитирующие поведение пользователя, вводящего данные. Эти паузы также называют «временем на раздумья». Они предназначены для как можно более реалистичной имитации поведения человека, которому требуется время, чтобы ввести данные и подумать, что делать дальше.

Ближе к концу производится запуск теста. Он моделирует действия пяти пользователей. Нагрузка увеличивается каждые 10 секунд — то есть каждые 10 секунд количество действующих имитаций пользователей увеличивается на единицу. Это помогает избежать внезапного создания сразу большой нагрузки на приложение. Такой сценарий маловероятен на практике и может вызвать отказ приложения, который никогда не случился бы в рабочем окружении.

В процессе тестирования измеряется время ответов и других событий на разных этапах. Результат для "POST user data" изображен на рис. 5.5. Сначала приложению требуется какое-то время, чтобы «разогреться». Зато потом время ответов остается постоянно низким, кроме единственного пика на графике.

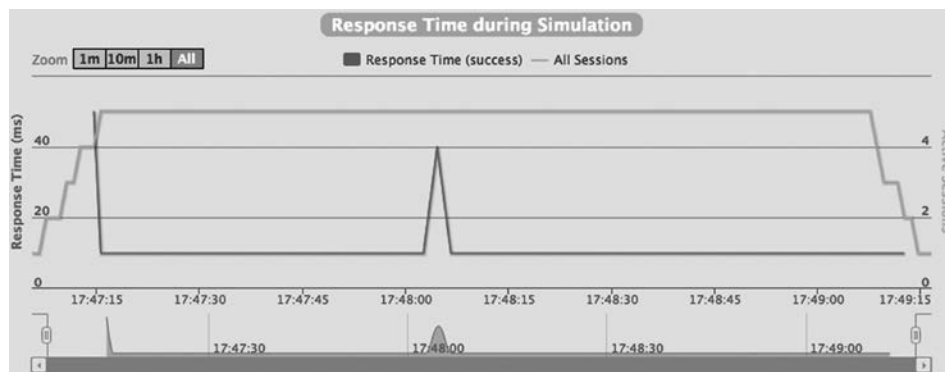


Рис. 5.5. Результаты тестирования пропускной способности

5.4.1. Различия между демонстрацией и практическим применением

Ради простоты и удобства демонстрации мы нарушили несколько важных правил. Во-первых, приложение и генератор нагрузки выполняются на одном компьютере. Всякий раз, когда в ходе тестирования требуется получить максимально реалистичную картину, генератор нагрузки и приложение должны запускаться на разных компьютерах. Иначе они будут конкурировать за ресурсы системы, и в результате получится картина, отличающаяся от характерной для рабочего окружения. Во-вторых, тест просто выводит результаты. То есть мы не определили условия, которые заставят тест потерпеть неудачу, если приложение покажет недостаточно высокую производительность.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Проект примера можно найти по адресу: <http://github.com/ewolff/user-registration-V2>. Загрузите его и выполните следующие инструкции.

- ▲ Установите Maven (см. <http://maven.apache.org/download.cgi>).
- ▲ Затем выполните команду `mvn install` в каталоге `user-registration`.
- ▲ После этого попробуйте выполнить тесты пропускной способности.
 - В подкаталоге `user-registration-capacitytest-gatling` запустите тестирование командой `mvn test`.
 - Результаты тестирования можно найти в файлах HTML, каталоге `target/gatling/results`.
 - Найдите Gatling Cheat Sheet в Интернете.
 - Найдите информацию о функциях проверки условий в Gatling. Как они работают в Gatling 2.0?
- ▲ Найдите тест в каталоге `src/test/scala`.
 - Измените тест так, чтобы он терпел неудачу в случае низкой производительности.
 - Для этого добавьте вызов, проверяющий условие. Выше вам уже предлагалось найти документацию с описанием этой возможности.
 - Условия можно определить в вызове `setUp`, по аналогии с `httpProtocol`.
 - Например, можно сравнить среднее время ответа на запрос "POST user data" с величиной 1 мс и, если оно превышает пороговое значение, завершать тест с ошибкой.
- ▲ Найдите документацию с описанием регистратора Gatling Recorder.
 - Загрузите Gatling и запустите регистратор.
 - Настройте веб-браузер на работу через прокси-сервер.
 - Запишите процесс поиска клиента. Это нормально, когда поиск завершается неудачей, потому что поиск выполняется всегда.
 - Запустите этот тест как часть тестирования пропускной способности. Для этого скопируйте код теста в проект.

- ▲ Выполните тестирование с разными тестовыми данными — например, используйте набор данных с 0, 1 и большим количеством результатов. Для этого воспользуйтесь объектом, подобным `emailFeeder` в примере выше, который использовался для создания уникальных адресов электронной почты.

5.5. Альтернативы инструменту Gatling

В настоящее время Gatling поддерживает только протокол HTTP. Однако вы можете писать свои расширения для поддержки других протоколов. Из-за того что возможности Gatling ограничиваются протоколом HTTP, этот инструмент оказывается не лучшим средством для реализации тестов пропускной способности через программный интерфейс.

5.5.1. Grinder

Grinder [2] — одна из возможных альтернатив инструменту Gatling. Этот инструмент позволяет реализовать тесты пропускной способности на Jython, варианте языка программирования Python, реализованном на Java. Поддерживается также альтернативный язык Clojure, напоминающий LISP, но выполняемый под управлением JVM. Соответственно, любые программные интерфейсы, доступные из Java, могут использоваться в Grinder для реализации тестов. То есть тестирование через программный интерфейс реализуется довольно просто. Кроме всего прочего, имеются предопределенные классы поддержки протоколов HTTP и JMS для тестирования приложений на серверах.

5.5.2. Apache JMeter

Еще один вариант — Apache JMeter [3]. Этот инструмент имеет графический редактор для тестов, которые помимо HTTP поддерживают другие протоколы, такие как JMS, SMTP и FTP. Также имеется возможность записать последовательность взаимодействий с веб-сайтом и затем воспроизвести ее. JMeter позволяет отображать результаты. Он — самый популярный и самый используемый инструмент в настоящее время. Существует коммерческое предложение для JMeter, которое называется

Blazemeter и выполняется в облачных компьютерах, предназначенных для нагрузочного тестирования. Конечно, оно работает только с общедоступными веб-сайтами.

5.5.3. Tsung

Другой интересный инструмент — Tsung [4], написанный на Erlang. Этот язык программирования специально создавался для создания распределенных систем. Кроме того, парадигма программирования на Erlang позволяет сгенерировать значительный объем сетевого трафика лишь несколькими потоками выполнения. Также относительно просто реализуется управление кластерами. То есть Erlang обладает идеальными характеристиками для реализации инструментов нагрузочного тестирования. Tsung поддерживает такие протоколы, как HTTP, WebDAV, SOAP, PostgreSQL, MySQL, LDAP и Jabber/XMPP; позволяет имитировать поведение пользователей и осуществлять мониторинг производительности системы.

5.5.4. Коммерческие решения

К числу коммерческих решений можно отнести HP LoadRunner [5] и Rational Performance Tester [6]. Кроме того, имеются поставщики коммерческих услуг, предлагающие возможность нагрузочного тестирования в Интернете. В этом случае тесты выполняются — как в случае с Blazemeter — в облаке, поэтому не требуется иметь собственные серверы или заниматься установкой и настройкой дополнительного программного обеспечения. Кроме того, оплачиваются только ресурсы, израсходованные за фактическое время тестирования. Такие предложения позволяют использовать большое количество компьютеров и, соответственно, создавать огромные нагрузки. Часто поставщики услуг предлагают также простые и эффективные средства анализа и оценки результатов тестирования. Кроме того, тесты используют веб-сайт в точности, как он использовался бы клиентами, то есть на рабочем аппаратном обеспечении с определенной топологией сети.

Spirent Blitz [7] — еще один пример поставщика услуг, который позволяет разрабатывать и выполнять комплексные нагрузочные испытания с помощью простого веб-сайта. Аналогичные возможности дает LoadStorm [8]. Здесь нагрузочные тесты могут быть реализованы путем записи взаимодействий в веб-браузере.

5.6. В заключение

Тестирование пропускной способности гарантирует поддержку приложением ожидаемого количества пользователей и достаточно высокое быстродействие. Методология непрерывного развертывания предлагает выполнять тесты пропускной способности после каждого изменения, чтобы постоянно иметь самую свежую информацию о производительности приложения. Важными условиями успеха являются определение обоснованных требований к производительности приложения и подготовка такого окружения тестирования, которое позволит сделать выводы о производительности в рабочем окружении.

Тесты могут взаимодействовать с приложением через специальный программный интерфейс или интерфейс пользователя — например, веб-интерфейс.

Пример с Gatling демонстрирует типичную процедуру тестирования, когда выполняется запись взаимодействий с веб-сайтом. Далее на основе этой записи реализуется соответствующий тест на предметно-ориентированном языке, который оценивает производительность приложения. Код теста можно корректировать. Его можно включить в конвейер непрерывного развертывания для измерения текущего уровня производительности при каждом запуске конвейера.

Альтернативами Gatling могут служить Grinder, где тесты записываются на языке Clojure или Jython, и Apache JMeter, где предусматривается возможность реализации тестов с применением графического построителя.

Ссылки

1. <http://gatling.io/>
2. <http://grinder.sourceforge.net/>
3. <https://jmeter.apache.org/>
4. <http://tsung.erlang-projects.org/>
5. <http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/>
6. <http://www-03.ibm.com/software/products/en/performance>
7. <https://www.blitz.io/>
8. <http://loadstorm.com/>

6

Исследовательское тестирование

6.1. Введение

До сих пор основное внимание уделялось автоматизированным тестам. Но в этой главе мы будем рассматривать ручное тестирование и в особенности тестирование, которое называют исследовательским. В разделе 6.2 рассказывается, где и с какой целью используются исследовательские тесты. В разделе 6.3 демонстрируются конкретные процедуры выполнения таких тестов. Исчерпывающее введение в эту тему также можно найти в книге «Explore It!» [1], где концепции этого вида тестирования объясняются более подробно.

6.1.1. Исследовательское тестирование: пример

В компании «Big Money Online Commerce Inc.», с которой мы познакомились в разделе П.2, накоплен большой опыт ручного тестирования — каждая новая версия подвергалась всестороннему тестированию вручную. Однако из-за автоматизации приемочных испытаний фокус ручных тестов сместился: теперь главной их целью является тщательная проверка новых функциональных возможностей. В этом контексте наиболее важным является понимание работы программного обеспечения с предметной точки зрения. Это — характерная особенность исследовательского тестирования, осуществляемого вручную. Исследовательское тестирование проводится также с целью оценить и усовершенствовать отдельные характеристики программного обеспечения, такие как удобство использования, защищенность, реакция на крайние ситуации.

6.2. Цель исследовательского тестирования

Главной целью методологии непрерывного развертывания является автоматизация тестов. Ручные тесты обходятся недешево, особенно когда

тестировать приходится часто. Каждый тест должен выполняться и анализироваться человеком. Автоматизированные тесты, с другой стороны, должны быть автоматизированы только один раз. Последующие запуски автоматизированных тестов практически ничего не стоят. Благодаря методологии непрерывного развертывания приложения часто развертываются и, соответственно, часто тестируются. Поэтому тестирование приходится автоматизировать, иначе его стоимость будет слишком высока.

6.2.1. Иногда ручное тестирование оказывается предпочтительнее

Как бы то ни было, в некоторых случаях ручное тестирование может оказаться предпочтительнее: когда вводятся новые возможности, специалисты в прикладной области должны сначала проверить их вручную. Чтобы суметь оценить новые возможности, тестировщики должны знать и понимать предметный контекст. Основой таких тестов могут служить требования или пользовательские истории. Кроме того, ручные тесты дают возможность еще раз критически изучить требования и выявить возможные ошибки. Естественно, этого нельзя достичь с использованием автоматизированных тестов, потому что они должны основываться на определенных требованиях.

Исследовательское тестирование основывается на том факте, что тестировщики часто обнаруживают проблемы, только когда отклоняются от сценария и самостоятельно изучают и исследуют приложение. При этом они руководствуются собственным опытом и знанием типовых уязвимостей приложений. Очевидно, что тесты этого вида нельзя автоматизировать — их может выполнить только опытный тестировщик.

Исследовательское тестирование может оказаться превосходным инструментом детального изучения особенностей, реализованных с ошибками, или где высока вероятность появления ошибок. Когда, например, конкретная функция или модуль чаще других вызывают проблемы в рабочем окружении, в ходе исследовательского тестирования можно особенно тщательно проанализировать эту область, чтобы выявить дополнительные ошибки.

6.2.2. Тестирование заказчиком

Иногда заказчик может пожелать протестировать новую версию, прежде чем принять ее. И снова в этом случае главное внимание уделяется новым возможностям. Если заказчик также тестирует давно реализованные функ-

ции, разработчики должны использовать это как возможность для повышения доверия к автоматизации. Например, разработчик может показать, как функции, которые заказчик оценивает вручную, тестируются в автоматическом режиме. Кроме того, разработчик может реализовать для заказчика приемочные тесты новых особенностей для заказчика, выполняющиеся автоматически.

Регрессионные тесты никогда не должны выполняться вручную, потому что это ведет к неоправданному увеличению затрат, когда требуется тестировать каждый релиз. С другой стороны, не всякое требование должно приводить к созданию автоматизированного теста. Вполне достаточно, если автоматизированные тесты дают разработчикам определенную уверенность в нормальной работоспособности новой версии. Для этого приложение должно быть протестировано до такой степени, что все случаи использования в тестах выполняются без ошибок.

6.2.3. Ручное тестирование нефункциональных требований

Исследовательскому тестированию обычно подвергаются новые функциональные возможности. Однако ручные тесты могут быть предпочтительнее автоматизированных, когда дело касается нефункциональных требований.

- *Тестирование удобства в использовании* позволяет оценить простоту пользования приложением. Очень легко выполняется человеком, но с большим трудом поддается автоматизации.
- То же верно для *оценки внешнего вида* и соответствия требованиям к оформлению. Часто человек способен сразу обнаружить ошибки, но автоматизировать этот процесс чрезвычайно сложно.
- Наконец, выявление *проблем безопасности* зачастую проще осуществить вручную, а не в виде автоматизированного процесса. В этой области часто применяются приемы обзора программного кода или проверки возможности вторжения, которые тоже весьма сложно автоматизировать.

6.3. Как это сделать?

Исследовательское тестирование в первую очередь направлено на проверку прикладной функциональности. То есть тестировщики должны оценить

соответствие приложения выдвинутым требованиям. Поддержку автоматизации рутинных операций при проведении тестирования могут оказать такие инструменты, как Selenium (см. главу 4, «Приемочные тесты»). Они записывают последовательность взаимодействий с веб-интерфейсом приложения и воспроизводят их снова и снова.

Вслед за этим тестировщики выполняют операции с приложением вручную. Такие сценарии автоматизации могут также послужить основой для более поздней, полной автоматизации тестирования.

6.3.1. Руководство по проведению тестирования

Планы тестирования, описывающие, что и как тестировать, безусловно, полезны сами по себе. Однако они также помогают определить момент, когда тесты достигнут определенного уровня зрелости и сложности и их автоматизация сможет дать определенные выгоды. Поэтому в исследовательском тестировании вместо планов используются уставы (см. раздел 6.3.6).

6.3.2. Автоматизированное окружение

Окружение, созданное автоматически, послужит хорошей основой для исследовательского тестирования. В этом окружении необходимо установить программное обеспечение и соответствующие тесты.

Исследовательскому тестированию должны подвергаться только версии, успешно преодолевшие все другие стадии — то есть тестирование, выполняющееся на этапе сохранения изменений, автоматизированные приемочные испытания и тестирование пропускной способности. Иначе усилия, потраченные на ручное тестирование, окажутся затраченными зря на проверку версии, имеющей сомнительное качество.

6.3.3. Демонстрационные примеры как основа

Демонстрационные примеры служат надежной основой для исследовательских тестов. В проектах, развиваемых с применением методик гибкой разработки, в конце итерации клиенту часто демонстрируются примеры, показывающие, как работают вновь добавленные возможности и какие выгоды они дают. Именно эти возможности также должны подвергаться исследовательскому тестированию. Поэтому одним из результатов иссле-

довательских испытаний может быть успешный прогон через такой демонстрационный пример.

6.3.4. Пример: приложение электронной коммерции

Предположим, что в приложении электронной коммерции была реализована возможность создания срочных заказов. В ходе исследовательского тестирования эксперт создает такой срочный заказ и проверяет, насколько правильно он обрабатывается. В дополнение к этому очевидному случаю тестировщик выясняет, что произойдет, если клиент попытается оформить срочный заказ на товар, недоступный в данный момент, или срок доставки которого превышает время выполнения срочного заказа. Аналогично может быть проверена функция отмены срочного заказа. При этом особое внимание уделяется процедурам, подверженным ошибкам, например обновлению состояния заказа. Благодаря знаниям бизнес-процессов, особенностей работы текущей версии системы и областей, где возможны ошибки, ручное тестирование может быть весьма эффективным. При этом, конечно, создается хорошая основа для автоматизации тестирования, которое впоследствии защитит приложение от регрессий.

Кроме того, в ходе исследовательского тестирования проверяется удобство использования и внешнее оформление с применением методов, установившихся в этой области. Здесь пригодятся опросы пользователей, а также запись взаимодействий с программным обеспечением с последующим анализом и оценкой.

Кроме перечисленного во время ручного тестирования может проверяться защищенность приложения — например, проверка возможности вторжения или обзор кода в поисках проблем с безопасностью. Аналогично вручную проверяются пропускная способность или производительность. Однако в последнем случае обычно требуется использовать генератор нагрузки, поэтому такие тесты трудно реализовать без автоматизации.

6.3.5. Бета-тестирование

Еще одна разновидность исследовательских тестов — бета-тесты. В этом случае программное обеспечение передается ограниченному кругу пользователей, которые, как предполагается, опробуют его и сообщат об обнаруженных ошибках и предложениях по улучшению. В некоторых случаях это

позволяет одновременно проверить, позволяет ли новая версия увеличить объем продаж.

6.3.6. Сеансовые тесты

Для структурирования исследовательского тестирования удобно использовать метод сеансовых тестов [2]. В этом случае тесты делятся на сеансы. Для каждого сеанса определяется его цель — виды ошибок или проблем, которые должны быть проверены. Сеанс можно сравнить с экспедицией в приложение. Цель экспедиции определяется уставом. Устав географической экспедиции может, например, определить такую цель, как исследование определенного ландшафта. Устав исследовательского тестирования аналогично определяет цели каждого сеанса. Устав может иметь следующую форму:

Исследовать ... (цель)
с помощью ... (инструмент),
чтобы определить ... (информация).

Эти три части определяют следующие аспекты.

- *Цель* — часть приложения, которая подвергается тестированию, — то есть определенная функция, требование или модуль.
- *Инструмент* определяет средства, используемые для тестирования. К инструментам можно отнести определенные наборы данных или программные средства.
- Наконец, *информация* определяет результат тестирования. Результатами могут быть выводы о защищенности, быстродействии или надежности. То есть во время исследовательского тестирования обязательно должны проверяться конкретные предметные требования и ошибки.

В контексте приложения регистрации клиента можно было бы определить такой устав исследовательского тестирования:

Исследовать приложение регистрации клиента
с помощью соответствующего набора данных,
чтобы убедиться в правильной интернационализации.

В ходе тестирования в соответствии с этим уставом проверяется возможность использования букв национальных алфавитов и других наборов сим-

волов, например корейских, японских или китайских. Возможно ли, что деление на имя и фамилию подходит не для всех стран? И какие символы фактически допустимы в адресах электронной почты?

Ниже приводится пример совершенно другого устава:

Исследовать приложение регистрации клиента
с помощью атак OWASP,
чтобы выявить пробелы в системе защиты.

OWASP¹ [3] — это коллекция наиболее известных уязвимостей в веб-приложениях. В данном случае целью исследовательского тестирования является оценка защищенности — соответствие нефункциональному требованию.

Оба устава имеют общее свойство, формулируемое универсальным способом. Ручные тесты необязательно должны быть полностью предопределены; допустимо сформулировать только цель и используемые методы. Тестировщики сами определяют конкретную реализацию сеанса.

Уставы могут определяться на основе требований или преследовать цель более тщательного исследования определенных рисков — например, выявление пробелов в системе защиты на ранних этапах.

Формальным результатом исследовательского тестирования является отчет о проведенном сеансе, содержащий, кроме устава, сведения о том, как выполнялось тестирование, какие ошибки были обнаружены и как долго длился сеанс. В заключение дается ретроспективный обзор сеанса. Сеанс длится всего несколько часов. Соответственно, ручное тестирование можно разбить на несколько разных сеансов с определенными целями и таким способом структурировать его.

Тестировщики имеют определенную свободу в выборе инструментов. В дополнение к использованию инструментов автоматизации они могут исследовать файлы журналов или параметры работы системы. Для этого могут очень пригодиться инструменты для мониторинга и анализа файлов журналов, представленные в этой книге (глава 8, «Эксплуатация»). Кроме того, исследовательские тесты могут опираться на пользовательские и программные интерфейсы или на веб-службы. Естественно, все это требует наличия технических знаний у тестировщиков. Такие исследования могут

¹ Open Web Application Security Project — открытый проект обеспечения безопасности веб-приложений. — *Примеч. пер.*

проводиться только специалистами, знакомыми с необходимыми инструментами. В особых случаях от тестировщиков может даже потребоваться умение писать собственное программное обеспечение.

В течение сеанса тестировщики исследуют программное обеспечение с применением разных приемов.

- Приложения имеют *переменные*. В зависимости от типа они могут хранить разные значения. Поэтому имеет смысл проверить разные вариации — например, использование китайских символов в имени пользователя.
- Порядок *взаимодействий и процедур* может изменяться. Это позволяет исследовать, что случится, если некоторая процедура выполняется пользователем не так, как предполагалось изначально.
- Кроме того, тестировщики могут исследовать *сущности и отношения между сущностями*. В ходе этого тестирования можно попробовать создать недопустимую сущность.
- Многие приложения имеют определенный набор состояний и определяют порядок переходов между ними. Здесь тестировщики проверяют, насколько разумно реализованы состояния или возникновение неожиданных состояний.

Эти приемы позволяют тестировщикам различать уставы и исследовать потенциальные слабости программного обеспечения.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ Реализуйте один из уставов, упомянутых выше, для тестирования приложения регистрации клиента.
- ▲ Какие другие уставы вы могли бы предложить для тестирования приложения регистрации клиента?

Взгляните на любой из своих проектов.

- ▲ Выберите определенную функцию. Какой устав для ее тестирования вы могли бы предложить?
- ▲ Какие нефункциональные требования (например, защищенность или производительность) являются особенно важными для приложения? Определите устав для их тестирования.

6.4. В заключение

Для большей ясности отметим еще раз: в проекте, который развивается с применением методологии непрерывного развертывания, не должно быть слишком много тестов, выполняемых вручную. Непрерывное развертывание предполагает частое развертывание и тестирование; на самом деле — очень частое. Поэтому тесты почти всегда имеет смысл автоматизировать. Однако не бывает правил без исключений: исследовательское тестирование новых возможностей в действительности возможно только вручную. Такое тестирование позволяет получить более полное представление об уязвимых аспектах приложения и реализовать автоматизированное тестирование для защиты от регрессий. То же верно для тестирования удобства в использовании и оформления, а также других нефункциональных требований, таких как проверка возможности вторжения. Исследовательское тестирование новых возможностей должно в конечном итоге приводить к созданию автоматизированных тестов. Кроме того, исследовательское тестирование может послужить хорошим способом более тщательного изучения некоторых аспектов приложения. В конце концов, автоматизированное тестирование не способно выявить абсолютно все ошибки.

Ссылки

1. Elisabeth Hendrickson: *Explore It! — Reduce Risk and Increase Confidence with Exploratory Testing*, Pragmatic Bookshelf, 2013, ISBN 978-1-93778-502-4.
2. <http://www.satisfice.com/sbtm/>
3. https://www.owasp.org/index.php/Main_Page

7

Развертывание — ввод в эксплуатацию

7.1. Введение

Развертывание в рабочем окружении — это просто еще один вид развертывания. В главе 2, «Подготовка инфраструктуры», уже обсуждались инструменты для развертывания. Однако ошибки во время ввода в эксплуатацию обходятся намного дороже ошибок во время развертывания в тестовом окружении. Поэтому в этой главе рассматриваются подходы, помогающие еще больше снизить риски, связанные с вводом в эксплуатацию. В разделе 7.2 рассказывается о необходимости иметь возможность отката и возврата к предыдущей версии программного обеспечения. В случае проблем можно немедленно развернуть (накатить) исправления, устраняющие ошибку (раздел 7.3). Еще одна возможность описывается в разделе 7.4: сине-зеленое развертывание, предусматривающее создание совершенно нового окружения для новой версии. Эта методика позволяет запустить новую версию, не останавливая старой, и затем, когда все будет проверено и готово, просто переключиться между ними. С другой стороны, канареечное развертывание (раздел 7.5), предусматривает развертывание программного обеспечения на ограниченном количестве серверов и только потом — на всех компьютерах. В разделе 7.6 демонстрируется использование методологии непрерывного развертывания для ввода в эксплуатацию каждого изменения в коде. Большинство подходов, представленных в этой главе, ориентировано на веб-приложения. В разделе 7.8 объясняется, как выполнять развертывание приложений других типов.

Базы данных — это особый случай развертывания в рабочем окружении. Эта тема подробно рассматривается в разделе 11.5, поэтому мы не будем касаться ее в этой главе.

7.1.1. Развертывание: пример

Первоначально сотрудники компании «Big Money Online Commerce Inc.», представленной в разделе П.2, предполагали, что благодаря автоматизации инфраструктуры развертывание в рабочем окружении не будет вызывать сложностей и риски, связанные с этим этапом, сведутся к минимуму. И действительно, так и получилось. Однако для развертывания новой версии все еще необходимо останавливать веб-сайт, и, соответственно, он на какое-то время становится недоступным. Кроме того, однажды развертывание все же завершилось неудачей из-за различий между рабочим и тестовым окружениями. Это нарушило нормальное функционирование рабочего окружения. Поэтому для обеспечения безопасного ввода программного обеспечения в эксплуатацию потребовалось предусмотреть дополнительные меры. Недостаточно просто обновить текущую версию программного обеспечения в рабочем окружении. Необходимо еще иметь возможность откатиться до предыдущей версии или избежать нарушения ритма нормальной работы иными способами. Такие меры будут представлены в этой главе.

7.2. Ввод в эксплуатацию и откат

Управление рисками во время ввода в эксплуатацию имеет особое значение. Самый очевидный способ минимизировать риски — реализовать возможность отката к предыдущей версии. Если в новой версии программного обеспечения обнаруживается проблема, ее можно заменить прежней, выполнив откат.

Если возникает необходимость прибегнуть к этой альтернативе, значит, действительно что-то пошло не так и приложение оказывается недоступным в этот момент. Поэтому очень важно, чтобы данная процедура работала безупречно. Следовательно, она должна быть тщательно протестирована, чтобы гарантировать восстановление рабочего окружения в экстренной ситуации.

7.2.1. Преимущества

Сама процедура не должна быть излишне сложной — в конце концов, прежняя версия приложения была в свое время успешно развернута, — и необходимые операции не должны существенно отличаться от развер-

тивания новой версии. Проблемы может вызвать только откат изменений в базе данных (см. разделы 2.9 и 11.5), поскольку изменение схемы реляционной базы данных, особенно при наличии гигантского объема информации, сопряжено с определенными сложностями и проблемами. Для выполнения отката не требуется ничего особенного, кроме наличия прежней версии и утвержденной процедуры, определяющей порядок возврата старой версии.

7.2.2. Недостатки

Процесс ввода в эксплуатацию старой версии невозможно всесторонне протестировать, так как его нельзя опробовать в рабочем окружении. Поэтому откат всегда связан с некоторым риском. Степень риска увеличивается, если в базе данных произошли какие-то изменения, поскольку потери данных следует избегать.

В ходе отката в рабочем окружении должна быть развернута старая версия. Это требует некоторого времени, поэтому приложение останется недоступным для пользователей на этот период. В зависимости от требований к доступности это может означать, что откат в принципе невозможен. Другая проблема — изменения в базе данных: такие изменения порой очень трудно откатить. Это увеличивает риск невозможности отката, усложнения процедуры и увеличения времени простоя. То есть во многих случаях откат возможен только теоретически, если не использовать специальные подходы к управлению базами данных, описанные в разделах 2.9 и 11.5.

Еще одна проблема: даже при том, что после отката приложение вновь становится доступным, поиск причин, вызвавших проблемы с новой версией, затруднен. Решающее значение для выявления источника проблемы зачастую имеют файлы данных и журналов на рабочих компьютерах — однако именно эта информация часто удаляется во время отката. Более того, из-за необходимости выполнить откат как можно быстрее важная информация может быть удалена в спешке или по ошибке. В этом случае единственный способ решить проблему — симитировать ситуацию в рабочем окружении. Однако эта стратегия имеет ограниченные шансы на успех: приложение уже было всесторонне протестировано в окружении, напоминающем рабочее, например при проведении приемочных испытаний. Когда ошибка возникает только при вводе в эксплуатацию, ее будет очень трудно воспроизвести в любом другом окружении.

7.3. Развертывание исправлений

Взамен отката можно выполнить развертывание исправлений — также известное как *накатывание заплат*. В этом случае новая версия программного обеспечения развертывается поверх ошибочной. Она устраняет обнаруженные ошибки. Конечно, эти исправления также должны тестироваться. Однако благодаря настроенному конвейеру непрерывного развертывания это не создает дополнительных расходов. Наконец, этот подход основан на возможности конвейера непрерывного развертывания быстро и эффективно развернуть изменения, чтобы как можно скорее исправить ошибку.

7.3.1. Преимущества

Поскольку изменения с исправлениями обычно имеют относительно небольшой размер, развертывание обновленной версии выполняется существенно быстрее. По трудоемкости операция развертывания исправлений сопоставима с операцией отката, но она проще сама по себе: выпуск можно произвести без процедуры отката. В случае наличия изменений в базе данных их откат может быть затруднен, тогда как накатывание исправлений обычно не требует отменять изменения в базе данных; это упрощает данную процедуру.

Кроме того, в данном подходе отсутствуют крайние случаи: процедура устранения проблем идентична процедуре развертывания новой версии — и в идеале этот процесс может выполняться несколько раз в день. Это избавляет от необходимости дополнительного тестирования крайних случаев.

7.3.2. Недостатки

В методике развертывания исправлений отсутствует простая процедура возврата к гарантированно рабочей версии. Конечно, изменения в коде легко отменяются с помощью системы управления версиями; однако не всегда очевидно, какая часть кода является действительным источником ошибки. А если еще и в базе данных произошли изменения, старый код может отказаться работать с обновленной базой данных. Если в конечном итоге ошибка окажется намного сложнее, решение придется принимать в условиях острой нехватки времени.

Однако эти риски можно снизить дополнительными мерами. В главе 11, «Непрерывное развертывание, DevOps и архитектура ПО», рассказывается, как удачно подобранная архитектура способна уменьшить негативные последствия отказа в компоненте и как лучше вводить в действие изменения в базе данных. Приемы, представленные в этой главе, также помогают уменьшить риски. В конце концов, возможность быстро накатить исправления сама по себе свидетельствует о том, что команда хорошо поработала над реализацией конвейера непрерывного развертывания. Разработчики уверены в своей готовности ввести в эксплуатацию новую версию и в своем конвейере непрерывного развертывания настолько, что в экстренной ситуации предпочитают накатить новую версию, чем откатиться к старой. В то же время появляется возможность упростить конвейер непрерывного развертывания за счет исключения процедуры отката.

7.4. Сине-зеленое развертывание

При использовании методики сине-зеленого развертывания (blue/green deployment) новая версия программного обеспечения устанавливается в отдельную систему. Чтобы окончательно ввести новую версию в эксплуатацию, достаточно просто выполнить переключение с текущего окружения на новое — например, перенастройкой маршрутизатора (рис. 7.1).

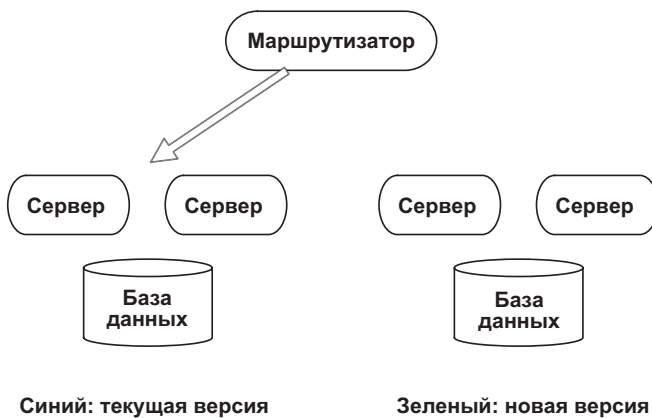


Рис. 7.1. Сине-зеленое развертывание

7.4.1. Преимущества

При таком подходе новая версия вводится в эксплуатацию практически без остановки приложения. Кроме того, появляется возможность сначала всесторонне протестировать приложение в новом окружении, проверив его производительность и функциональность. Если после ввода в эксплуатацию обнаружатся какие-то проблемы, возврат к предыдущей рабочей версии также выполняется очень просто; достаточно лишь перенастроить маршрутизатор и переключиться на старое окружение.

7.4.2. Недостатки

Очевидным недостатком этого подхода является большое потребление ресурсов: одновременно должно существовать два окружения и оба — мощные, чтобы справляться с рабочими нагрузками. То есть каждый компьютер, необходимый для работы, должен существовать в двух экземплярах. Эта проблема имеет несколько решений: использование общедоступных облачных услуг существенно снижает остроту проблемы. Второе окружение должно существовать лишь ограниченный период времени, пока новое развертывается и тестируется. Когда новая версия покажет, что надежно справляется с работой, старое окружение можно остановить. Поскольку плата за облачные ресурсы взимается только за периоды, когда они используются, выбор такого решения существенно уменьшит расходы по сравнению с тем, когда компания организует свой вычислительный центр, обеспечивающий постоянную доступность ресурсов.

Однако в обычных, не облачных, окружениях тоже можно уменьшить потребность в ресурсах. Например, вторым рабочим окружением может служить окружение для обкатки. В таком окружении выполняются последние тесты перед вводом новой версии в эксплуатацию. Следовательно, оно должно максимально точно соответствовать рабочему окружению. Для перехода от обкатки к эксплуатации остается только перенастроить приложение. Например, заменить тестовые данные фактическими рабочими и организовать вызов сторонних систем вместо их имитаций. Конечно, для предотвращения ошибок процесс превращения опробуемой версии в рабочую должен быть автоматизирован.

Кроме того, новую версию программного обеспечения можно установить на каждый компьютер рядом с рабочей. Обе версии должны использовать разные порты для отделения друг от друга. Однако такой подход осложня-

ет настройку. Кроме того, для отдельных серверов необходимо предусмотреть возможность поддержки двух версий одновременно.

В этом сценарии снова возникает проблема с базами данных, связанная с необходимостью синхронизации обоих окружений. Одно из возможных решений — ограничить доступ к базе данных разрешением только для чтения и тем самым запретить любые изменения на некоторый срок. Другие альтернативы описываются в разделах 2.9 и 11.5. Но в отсутствие возможности обезопасить операции с базами данных метод сине-зеленого развертывания оказывается ничуть не лучше методики на основе отката.

7.5. Канареечное развертывание

Канареечное развертывание (рис. 7.2) — еще один способ уменьшить риски, связанные с вводом новой версии в эксплуатацию. В данном случае новая версия сначала развертывается только на одном или нескольких серверах в кластере. Эти серверы настраиваются так, что изначально не способны отвечать на запросы пользователей. Это позволяет сначала проверить их работу под уменьшенной нагрузкой. Если программное обеспечение выдержит испытание, его можно постепенно разворачивать на все большем количестве серверов, пока все серверы не будут переведены на новую версию.



Рис. 7.2. Канареечное развертывание

Этот подход получил свое название в честь стратегии, использовавшейся шахтерами в угледобывающих шахтах. Шахтеры брали с собой в шахту клетки с канарейками, потому что эти птицы очень чувствительны к рудничному газу: если канарейки начинали беспокоиться или теряли сознание, это говорило о том, что пора покинуть шахту. Стратегия канареечного

развертывания действует аналогично: если в новом программном обеспечении, развернутом на нескольких первых серверах, обнаруживается проблема, его дальнейшее развертывание прекращается. То есть такой подход напоминает систему раннего предупреждения.

7.5.1. Преимущества

Этот подход позволяет развернуть новую версию без остановки приложения (развертывание с нулевым временем остановки). В то же время эта стратегия не требует большого количества дополнительных ресурсов. Если развертывание выполнять в периоды снижения нагрузки, некоторые серверы можно временно исключить из работы для установки новой версии и не опасаться, что после включения они сразу начнут испытывать большие нагрузки. Это также дает возможность протестировать приложение в рабочем окружении, не создавая проблем для пользователей. В ходе такого тестирования приложение можно проверить на наличие проблем в прикладной логике и оценить его производительность в рабочем окружении.

Естественно, возврат к старой версии в этом контексте выполняется очень просто: достаточно переустановить старую версию лишь на одном или нескольких серверах. Это не должно вызвать проблем, особенно при наличии автоматизированной процедуры установки. Как вариант, серверы с новой версией можно остановить и в автоматическом режиме подготовить серверы со старой версией.

7.5.2. Недостатки

Этот подход безоговорочно требует, чтобы база данных и сторонние системы поддерживали обе версии программного обеспечения. Это усложняет задачу. Поэтому в любой момент в рабочем окружении должно действовать не более двух версий. Наличие большего количества версий бессмысленно и влечет неоправданное усложнение. Единственной целью должен быть пошаговый перевод всего рабочего окружения на новую версию. Единственная причина присутствия сразу двух версий в рабочем окружении — минимизация рисков, связанных с вводом в эксплуатацию. Постоянное присутствие двух версий в рабочем окружении может порождать самые разные проблемы — например, ошибки всегда должны исправляться сразу в двух версиях, и для каждой версии должен быть настроен свой конвей-

ер непрерывного развертывания. Поэтому рабочее окружение не должно пребывать в таком состоянии дольше нескольких часов. Если новая версия окажется непригодной для эксплуатации, ее следует быстро удалить из рабочего окружения; с другой стороны, если версия хорошо показала себя, можно смело развернуть ее на всех остальных серверах.

7.6. Автоматическое развертывание

Если конвейер непрерывного развертывания полностью автоматизирован, можно организовать автоматическое развертывание приложения полностью в автоматическом режиме после каждого изменения. Этот подход радикально отличается от классических методов, и на первый взгляд кажется, что его очень трудно реализовать. В конце концов, методики сине-зеленого или канареечного развертывания придуманы не просто так, а чтобы гарантировать невозможность появления проблем в рабочем окружении или хотя бы быстрое их исправление — желательно так, чтобы они остались не замеченными пользователями.

Тем не менее при автоматическом развертывании также можно использовать такие стратегии, как канареечное развертывание, устанавливая новую версию приложения сначала на ограниченное количество серверов, прежде чем открыть его для всех пользователей. Если производится автоматическое развертывание каждого изменения, эти изменения оказываются небольшими, и риск появления проблем снижается. Очевидно, что чем меньше изменений, тем ниже вероятность появления проблем, вызванных этими изменениями. Такое развертывание по своим масштабам не может сравниться с классическим подходом к выпуску новых версий. Если реализовать архитектурные подходы, описанные в главе 11, может даже получиться так, что бо́льшая часть приложения вообще не будет меняться, а это способствует дальнейшему снижению рисков. Дополнительные меры помогут уменьшить риски еще больше. Например, развертывание может быть разрешено только в рабочие часы, а информация из системы мониторинга будет немедленно передаваться разработчикам для быстрого обнаружения проблем.

Очевидно, что автоматическое развертывание может быть реализовано только при наличии механизма переключения (раздел 11.7): первоначально новые функции отключаются в рабочем окружении и активируются только после их полной реализации, тестирования и отладки. Без такого

механизма отключения новых функций невозможно надежно обеспечить непрерывный ввод в эксплуатацию новых версий. Поскольку ручное тестирование в основном нацелено на проверку как раз новых функций, этот этап можно пропустить, если это допустимо.

7.6.1. Преимущества

Еще одно преимущество автоматического развертывания — дальнейшее уменьшение времени, необходимого на устранение проблем в рабочем окружении: если обнаруживается проблема, достаточно просто сгенерировать новую версию программного обеспечения (накатить исправления, см. раздел 7.3). Эта новая версия автоматически будет развернута, пройдя конвейер непрерывного развертывания. В отличие от типичной процедуры наложения исправлений, которая обычно выполняется в обход официального процесса, в данном случае используется обычная, стандартная процедура. Поскольку каждое изменение невелико, они относительно быстро переносятся в рабочее окружение. Поэтому данная модель не требует особого подхода к наложению исправлений. Кроме того, появляется возможность сэкономить на реализации процедуры отката: когда каждая новая версия содержит небольшой объем изменений, проще развернуть новую версию с исправлениями, чем выполнять откат. Это особенно верно при использовании дополнительных мер защиты качества, таких как канареечное развертывание.

Автоматическое развертывание увеличивает ответственность разработчиков, поскольку в рабочее окружение переносится каждое изменение. Это может положительно сказаться на качестве: разработчики должны убедиться, что новая версия не вызовет никаких проблем в рабочем окружении. Поэтому они с особым тщанием будут подходить к реализации конвейера непрерывного развертывания и включать в него все необходимые меры защиты, максимально препятствующие просачиванию ошибок в рабочее окружение. Кроме того, этот подход влечет необходимость оптимизации архитектуры программного обеспечения, чтобы при каждом обновлении передавались минимально возможные фрагменты, что также способствует повышению качества. Однако это также означает, что выбор архитектуры, не оптимизированной для автоматического развертывания, может сделать этот подход практически невозможным.

С упрощением переноса изменений в рабочее окружение увеличивается гибкость. Новую функцию можно быстро внедрить в эксплуатацию и, как следствие, быстро получить отзывы пользователей и оценить, как

эта функция ведет себя в рабочем окружении. Наконец, функцию можно быстро изменить или отключить в следующей версии.

7.6.2. Недостатки

Для надежной реализации этого подхода требуется максимально оптимизировать конвейер непрерывного развертывания. Это требует серьезных усилий. Кроме того, требуется надлежащим образом оптимизировать архитектуру программного обеспечения. Поэтому данный подход потребует значительных усилий, если проект изначально не был ориентирован на поддержку данной стратегии. Конечно, автоматическое развертывание можно реализовать только при достаточно высоком качестве конвейера непрерывного развертывания, в противном случае этот подход может легко привести к серьезным проблемам.

Кроме того, подход основывается на доверии: передать изменения в рабочее окружение может любой разработчик. Если в конвейере присутствует этап ручного тестирования, необходимо хотя бы самое элементарное ручное вмешательство. Пропуск этого этапа может оказаться проблематичным в некоторых окружениях с особенно строгими ограничительными требованиями. Часто выпуск должен выполняться людьми, не связанными с изменениями в коде или вообще работающими в другом подразделении. Однако это требование можно объединить с автоматическим развертыванием путем интеграции в конвейер принципов двойного контроля. Это снижает чувство ответственности у разработчиков, поскольку те осознают, что впереди еще имеется этап тестирования, на котором должны выявиться потенциальные ошибки. Развертывание непосредственно в рабочем окружении означает для разработчиков, что они сами несут ответственность за качество кода, и это чувство ответственности может способствовать повышению качества.

7.7. Виртуализация

Конечно, должна иметься возможность установки серверов в рабочем окружении. Для этого на сервере должны запускаться новые виртуальные машины, и на них должно устанавливаться необходимое программное обеспечение. Ниже перечисляются технологии, упрощающие этот процесс.

- *VMware* [1] предлагает комплексные коммерческие решения для организации даже очень больших серверных инфраструктур. Многие пред-

приятия применяют эти технологии, на практике доказавшие свою эффективность. В дополнение к гипервизору ESX для виртуализации отдельных серверов в линейке продуктов VMware имеется также решение vSphere, предназначенное для управления сложными инфраструктурами. Спектр продуктов дополняют инструменты аварийного восстановления и поддержки виртуальных сетей. Кроме того, VMware предлагает готовые решения для организации закрытого облака.

- *OpenNebula* [2] — решение с открытым исходным кодом, обладающее схожими возможностями. Кроме всего прочего, предлагает тот же программный интерфейс, что и Amazon Cloud, благодаря чему вы сможете дополнить ресурсы своего вычислительного центра ресурсами общедоступного облака. Решение OpenNebula разрабатывается компанией OpenNebula Systems и имеет целью предоставить предприятиям простое в использовании окружение.
- *OpenStack* [3] — еще одно решение с открытым исходным кодом. Дальнейшая разработка OpenStack поддерживается многими коммерческими поставщиками, которые создают и предлагают собственные дистрибутивы OpenStack. Как результат эта технология получила относительно широкое распространение. OpenStack состоит из множества отдельных служб для управления хранилищами, вычислительными ресурсами и сетями. Из-за большого количества служб установка может превратиться в сложную задачу. Подобно OpenNebula, OpenStack может работать с разными гипервизорами виртуализации, такими как Xen, KVM или ESX.

В числе дополнительных, облачных решений можно назвать Eucalyptus [4] и CloudStack [5].

На данный момент существуют также решения, специализированные для Docker (см. раздел 2.5), которые могут напрямую выполнять контейнеры Docker. В других решениях виртуализации, описанных выше, также имеются рудименты прямой поддержки Docker.

То есть виртуальные машины можно с успехом использовать в рабочем окружении. Как уже отмечалось в главе 2, «Подготовка инфраструктуры», существуют разные технологии установки приложений в эти виртуальные машины. В идеальном случае после запуска виртуальной машины VM, программное обеспечение сразу же устанавливается в виртуальное окружение. Порядок настройки таких окружений не описывается подробно в этой книге, поскольку она служит лишь введением в имеющиеся технологии автоматизации установки приложений.

7.7.1. Физические хосты

Описанные подходы в принципе можно применить к рабочим окружениям на физических хостах. В конце концов, для большинства технологий не важно, где установлено приложение — на виртуальных или физических хостах. Кроме того, гибкость, предлагаемая виртуализацией, не является решающим преимуществом для организации рабочего окружения: да, она удобна для создания тестовых окружений — например, чтобы установить несколько окружений с разными версиями или для разных задач — но рабочее окружение существует в единственном числе. Однако такие подходы, как канареечное, сине-зеленое и автоматическое развертывание, намного проще реализовать с использованием виртуальных машин.

7.8. Вне круга веб-приложений

Подходы, представленные к данному моменту, ориентированы на развертывание программного обеспечения на серверах. Это очень удобно, когда серверы контролируются организацией, которая также разрабатывает программное обеспечение, потому что существенно упрощает прямое развертывание. Однако существует также программное обеспечение, действующее иначе. Обычно оно передается клиенту и устанавливается им самостоятельно. В таком случае трудно применить подходы, описанные в этой главе. Один из примеров — мобильные приложения, которые распространяются только через интернет-магазин.

В таких сценариях возможны следующие подходы.

- Развертывание на компьютере клиента можно автоматизировать, например, с помощью подходящего сервера обновлений. Теоретически это позволяет развернуть новые возможности в любой момент. В интернет-магазинах приложений для мобильных устройств уже поддерживается такая возможность. Однако с тем же успехом аналогичную возможность можно реализовать для настольных и серверных приложений. В этом случае на стороне клиента можно развернуть любую версию. Чтобы избежать недовольства пользователей, важно позаботиться, чтобы не выкатить слишком много изменений сразу. Кроме того, процесс обновления должен быть максимально простым и надежным. Конечно, это требует тщательного тестирования процесса обновления.

- Также можно реализовать вариант канареечного развертывания: большинство приложений имеют круг особенно опытных пользователей, которые интенсивно работают с этими приложениями и часто дают отзывы об их работе. Таким пользователям можно чаще передавать новые версии — например, посредством автоматизированного канала — и просить поделиться своим мнением. Это позволит добиться высокой частоты выпуска новых версий, по крайней мере для некоторых пользователей. Они, в свою очередь, получают возможность напрямую влиять на разработку продукта и пользоваться самыми новыми возможностями.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Выберите проект, который вы хорошо знаете.

- ▲ Позволяет ли он откатить неудачную попытку развертывания?
- ▲ Автоматизированы ли необходимые процессы?
- ▲ Протестированы ли процессы?
- ▲ Был ли в практике случай успешного выполнения отката?
- ▲ Действительно ли откат помогает решить проблемы с выпуском новой версии?
- ▲ Возможно ли использовать прием сине-зеленого развертывания?
- ▲ Как может быть организована необходимая инфраструктура?
- ▲ Как решаются проблемы с базами данных и сторонними системами при переходе на новую версию?
- ▲ Возможно ли использовать прием канареечного развертывания?
- ▲ Как решаются проблемы с базами данных и сторонними системами для поддержки обеих — старой и новой — версий программного обеспечения?
- ▲ При каких условиях можно было бы применить автоматическое развертывание?
- ▲ Необходимо ли для этого изменить архитектуру проекта? См. также главу 11.
- ▲ Какие усовершенствования можно было бы добавить в конвейер непрерывного развертывания?

7.9. В заключение

Благодаря применению методологии непрерывного развертывания процесс ввода программного обеспечения в эксплуатацию технически является все тем же процессом, который уже используется для подготовки разного рода тестовых окружений. Но поскольку риски при развертывании в рабочем окружении выше, необходимо иметь возможность отката до стабильной версии приложения. Однако это не всегда просто реализовать. Поэтому нередко лучшей альтернативой оказываются канареечное или синее-зеленое развертывание. Автоматическое развертывание является наиболее радикальным решением: в рабочее окружение автоматически переносится каждое изменение. Описанные подходы также применяются к приложениям, устанавливаемым локально, на компьютерах клиентов. Однако в этом случае придется преодолеть дополнительные препятствия и, если необходимо, сократить частоту выполнения процедуры обновления, чтобы не вынуждать клиента обновляться слишком часто. Это противоречит желанию наладить более частый выпуск новых версий для минимизации рисков.

Ссылки

1. <http://www.vmware.com/>
2. <https://opennebula.org/>
3. <https://www.openstack.org/>
4. <http://www8.hp.com/us/en/cloud/helion-eucalyptus.html>
5. <http://cloudstack.apache.org/>

8

Эксплуатация

8.1. Введение

Основное внимание в этой главе уделяется обсуждению приемов эксплуатации приложений в рабочем окружении. В разделе 8.2 описываются проблемы, возникающие в период эксплуатации. Файлы журналов являются важным источником информации о состоянии дел в рабочем окружении. В разделе 8.3 демонстрируется, как использовать файлы журналов для записи и анализа информации о работе приложения. Раздел 8.4 демонстрирует практический пример анализа файлов журналов. С этой целью используется комплекс инструментов ELK (Elasticsearch, Logstash и Kibana). Elasticsearch сохраняет информацию из файлов журналов, Logstash собирает и анализирует сведения, а Kibana предоставляет средства визуализации результатов анализа. Также приводится пример окружения для ваших собственных исследований. Это окружение генерируется полностью автоматически, с помощью Docker и Vagrant. Разумеется, существуют также альтернативные решения (раздел 8.5). В разделе 8.6 обсуждаются расширенные вопросы ведения журналов.

Кроме журналирования, сведения о работе приложений можно также получать, используя технологии мониторинга (раздел 8.7) — в этом случае основное внимание уделяется учету числовых показателей. В разделе 8.8 демонстрируется, как можно визуализировать измеренные показатели с помощью Graphite. И снова, с использованием примера приложения, в разделе 8.9 рассказывается, как именно работает мониторинг. Здесь вам будет предложено для экспериментов готовое окружение с Graphite, созданное на основе Docker и Vagrant. Альтернативные решения для мониторинга будут представлены в разделе 8.10. В заключение в разделе 8.11 описываются некоторые дополнительные проблемы, связанные с эксплуатацией приложений.

8.1.1. Эксплуатация — пример

В разделе П.2 вы познакомились с компанией «Big Money Online Commerce Inc.» и узнали, какие проблемы возникли при вводе новой версии в эксплуатацию: регистрация новых клиентов на время отключилась. В первый момент никто не заметил этого, хотя эта проблема напрямую и сильно повлияла на успех приложения на рынке. Меры, описанные в этой главе, помогут получить более полное представление о работе приложения во время эксплуатации и избежать подобных ситуаций.

8.2. Проблемы в период эксплуатации

Для методологии непрерывного развертывания важна не только установка программного обеспечения в рабочее окружение. Не менее важно получение обратной связи во время эксплуатации. Вообще приложение должно функционировать в рабочем окружении без каких-либо ошибок. Однако чтобы иметь возможность вмешиваться в случае неизбежных ошибок и выяснять направления дальнейшего развития, необходимо получать данные из рабочего окружения. Следовательно, должен осуществляться мониторинг приложения. Для этого необходимо принять во внимание реализацию следующих аспектов.

- Инфраструктура — серверы, сетевое и прочее оборудование — должна подвергаться мониторингу. Это необходимо для выявления технических проблем, таких как выход из строя или перегрузка технических компонентов, и для вывода сообщений о проблемах, если это необходимо, чтобы дать возможность их устранить. Это основа классического подхода к организации эксплуатации. Методология непрерывного развертывания мало влияет на классический мониторинг, поэтому данный аспект не будет обсуждаться далее.
- Приложение также может поставлять данные о технических проблемах и ошибках. Это позволяет реагировать на появление конкретных ошибок в приложении. Данный аспект реализуется с применением специализированных инструментов или путем интеграции приложения с существующей инфраструктурой эксплуатации. Приложение должно поставлять соответствующую информацию так, чтобы используемые инструменты могли отображать и анализировать ее.

- Наконец, приложение также может поставлять данные о происходящем с прикладной точки зрения. Инструменты, используемые для анализа технических проблем, также могут обрабатывать подобную информацию и позволять делать интересные выводы о событиях, происходящих вокруг приложения. Однако иногда целесообразнее использовать специализированные инструменты — например, для анализа ссылок, выбираемых пользователями.

Целью мониторинга действующих серверов является не только запись и обработка информации о технических проблемах приложения, но также получение прикладных данных. С этой точки зрения применение методологии непрерывного развертывания и сопутствующих инструментов выглядит особенно интересным. Поэтому именно на этом мы сосредоточим основное внимание в данной главе.

В общем случае экспортирование прикладных данных из приложения для анализа с прикладной точки зрения не является чем-то новым. Кроме мониторинга данные могут экспортироваться для анализа, например, в различные хранилища — реляционные базы данных, специализирующиеся на хранении больших объемов данных и получении статистической информации на их основе. Существуют также соответствующие продукты, поддерживающие создание и анализ статистик. Аналогичный подход получил название «Большие данные» (Big Data): этот общий термин обозначает круг относительно новых технологий, предназначенных для обработки гигантских объемов данных, таких как базы данных NoSQL. В их основе лежит иная модель, отличная от реляционной, благодаря чему обеспечивается экономически эффективная масштабируемость: вместо больших серверов можно приобрести множество более дешевых и менее производительных серверов. На техническом языке это называется «горизонтальное масштабирование». В разделе 8.3 мы познакомимся с Elasticsearch — инструментом для сохранения и анализа файлов журналов. Elasticsearch — это поисковый механизм, но он поддерживает горизонтальное масштабирование, подобно базам данных NoSQL, и помимо поисковых индексов сохраняет также исходные данные. В этом отношении он очень похож на базу данных NoSQL.

Однако стратегии, лежащие в основе этих инструментов, отличаются от стратегий, используемых инструментами, представленными в этой главе. Они в основном опираются на экспорт информации из базы данных. Для этого используются так называемые инструменты извлечения-преобразования-загрузки (Extract-Transform-Load, ETL). Данные извлекаются из

приложения и преобразуются в структуры данных, пригодные для последующего анализа. В ходе этого процесса данные могут агрегироваться — объем продаж для одного клиента может быть не важен для статистики, но суммы продаж по определенным регионам или продуктам в день могут быть очень интересны. В заключение эта информация загружается в базу данных для анализа.

Существует множество альтернатив инструментам мониторинга и журналирования, представленным в этой главе, особенно интересных для анализа прикладных данных. Однако мы познакомимся только с инструментами, которые записывают данные, касающиеся эксплуатации, которые, впрочем, могут также использоваться для получения прикладных данных.

8.3. Файлы журналов

Файлы журналов — очень простой и на первый взгляд не впечатляющий способ записи информации о работе приложения. Информация записывается в файл и сопровождается некоторыми дополнительными сведениями, такими как метка времени. Однако этот подход имеет ряд преимуществ.

- Запись в файлы журналов осуществляется легко и просто — ее можно организовать с применением любой технологии. Для большинства языков программирования существует богатый выбор библиотек, поддерживающих операции с файлами журналов.
- Для сохранения данных не требуется прилагать больших усилий.
- Поскольку приложения только добавляют информацию в файлы журналов, эта операция практически не влияет на их производительность — как известно, операция записи является самой эффективной операцией с файлами.
- Большие объемы информации могут обрабатываться, например, с применением ротации файлов журналов: если размер файла превышает некоторый установленный порог, создается новый файл, куда продолжает записываться информация. Спустя какое-то время данные сжимаются или удаляются.
- Файлы легко искать и анализировать. Такие инструменты, как Grep, позволяют фильтровать информацию из файлов. Можно быстро написать

несколько сценариев, извлекающих необходимую информацию, и таким способом, шаг за шагом, проанализировать поведение приложения.

- Данные в журнале легко читаются и интерпретируются людьми, возможно, с применением очень простых инструментов. То есть анализ журналов не требует серьезных усилий. Однако это также означает, что информация из журналов обычно интерпретируется человеком. Поэтому она должна иметь представление, максимально простое для понимания. Когда проблема в рабочем окружении вызывает сбой ночью, не следует ожидать, что человек, обслуживающий систему, будет иметь необходимое время и достаточную концентрацию внимания, чтобы разгадать запутанные сообщения и их последовательности.
- Файлы журналов не зависят от конкретной технологии. Информация, записанная приложением в журнал, может анализироваться с применением самых разных технологий — отсутствует зависимость от какого-то одного решения мониторинга.

8.3.1. Что следует журналировать?

В файлы журналов можно записывать разные виды информации. Однако не все они одинаково важны. Поэтому системы журналирования обычно предлагают поддержку разграничения информации по уровням важности. Это позволяет отбрасывать сообщения, которые не должны попадать в файл журнала. Такой подход помогает минимизировать объем данных и влияние на производительность в рабочих окружениях и, напротив, сохранять всю информацию, которая поможет идентифицировать проблему в тестовых окружениях.

Проще говоря, поддерживаются уровни: *Error* — для критических ошибок, *Info* — для информационных сообщений и *Debug* — для подробных сообщений, адресованных разработчикам. Если система журналирования настроена на запись информации с меткой «Info», данные с более высоким приоритетом, таким как «Error», также будут записываться в журнал.

Существуют следующие эмпирические правила использования уровней журналирования.

- Возникающие ошибки всегда должны записываться в файл журнала. В идеале запись должна содержать всю информацию, необходимую для анализа ошибочной ситуации. Ошибки должны сохраняться с уровнем журналирования, таким как *Error*. Этот уровень необходимо зарезерви-

ровать для настоящих ошибок — во время нормального функционирования системы записи этого типа не должны появляться в файле журнала. Это гарантирует привлечение достаточного внимания к записям с типом *Error*. Если события с уровнем *Error* будут постоянно появляться в журнале в отсутствие настоящих ошибок, действительно важная ошибка может оказаться замеченной и исправленной с опозданием. Более того, в каждом окружении информация с этим уровнем обязательно должна сохраняться в файл журнала, чтобы ошибки не оставались без должного внимания. Журналирование ошибок чрезвычайно важно. Только когда каждая ошибка в рабочем окружении регистрируется в журнале и сопровождается достаточным объемом информации, есть шанс быстро обнаружить ошибку и исправить ее. В конце концов, в рабочем окружении нельзя использовать отладчик или другие инструменты для анализа ошибок, а даже если и было бы можно, ошибку еще необходимо воспроизвести.

- Файлы журналов также используются для сбора статистики во время нормальной эксплуатации приложения. Это реализуется добавлением записи с определенным уровнем важности, таким как *Info*, содержащей только информацию об успешно выполненных прикладных процедурах. Рабочие системы должны записывать такую информацию в файл журнала, если позднее предполагается анализировать ее.
- Наконец, подробную информацию с уровнем журналирования, таким как *Debug*, можно записывать в журнал для упрощения идентификации ошибок. Этот уровень журналирования должен активироваться только во время исследования ошибок и, в идеале, только в исследуемой части системы. Соответственно, этот уровень обычно отключается в рабочих окружениях.

При таком подходе файлы журналов легко могут использоваться для нужд исследования ошибок и мониторинга.

Сообщения в журнале могут также сопровождаться уникальным кодом. Это позволит быстро определить, где было сгенерировано то или иное сообщение. Кроме того, в руководстве или в другой документации, например в Вики, для каждого кода предоставляется дополнительная информация, описывающая возможные меры устранения ошибок. Для автоматизации таких мер используются соответствующие инструменты.

Однако при использовании большого количества серверов очень непросто заниматься исследованием файлов журналов, разбросанных по разным

серверам. Это требует организации согласованного доступа к ним. Кроме того, простые инструменты анализа файлов журналов имеют свой предел в отношении объема данных для исследования. В какой-то момент окажется недостаточно простого чтения данных от начала в поисках информации, как это делают такие инструменты, как Grep.

8.3.2. Инструменты для обработки файлов журналов

К счастью, существуют инструменты, специально предназначенные для таких задач. В сущности, они решают три основные задачи.

- Извлечение данных из разных файлов журналов, которые могут находиться на разных серверах в сети.
- Парсинг файлов: файлы журналов содержат такую информацию, как имя сервера, уровень журналирования или компонент, выполнивший запись. Помимо полнотекстового поиска должна поддерживаться возможность поиска по определенному полю.
- Результаты обработки журналов должны сохраняться так, чтобы обеспечить максимально эффективный поиск. Также должна существовать возможность осуществлять поиск по большим объемам данных.
- Наконец, должна иметься возможность анализировать данные и, соответственно, идентифицировать ошибки или анализировать поведение клиента.

ELK: Elasticsearch, Logstash, Kibana

Elasticsearch, Logstash и Kibana могут использоваться для следующих задач.

- Logstash [1] позволяет анализировать файлы и извлекать их из серверов в сети. Logstash — очень мощный инструмент. Он читает, изменяет или фильтрует данные и, наконец, записывает их. Кроме чтения файлов журналов и сохранения их в Elasticsearch поддерживаются также другие инструменты и механизмы ввода/вывода, такие как очереди сообщений или базы данных. Наконец, данные могут также анализироваться, например, по отметкам времени или отдельным полям.
- Elasticsearch [2] сохраняет данные и предоставляет их для анализа. Elasticsearch выполняет не только полнотекстовый поиск, но также поиск внутри структурированных данных и сохраняет результаты в хра-

нилище, например в базе данных. Наконец, Elasticsearch предлагает статистические функции, помогающие анализировать данные. Поисковый механизм в Elasticsearch оптимизирован для максимальной скорости отклика, что позволяет анализировать данные практически в интерактивном режиме.

- Kibana [3] — это веб-приложение для анализа данных из Elasticsearch. Кроме простых запросов также поддерживается статистический анализ.

Представленный комплекс инструментов известен под названием «стек ELK» (Elasticsearch, Logstash, Kibana) (рис. 8.1).

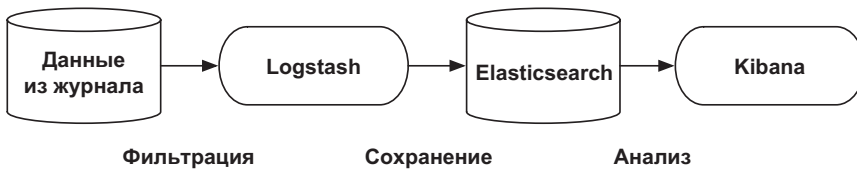


Рис. 8.1. Обработка журналов с применением стека ELK

8.3.3. Журналирование в примере приложения

Во-первых, код примера приложения необходимо изменить так, чтобы важная информация в конечном счете оказывалась в файле журнала. Для Java имеется много различных библиотек журналирования — в примере приложения используется Apache Commons Logging [4]. Фактическая запись в файлы за кулисами осуществляет Logback [5].

Приложение записывает в журнал информацию двух видов.

- В случае ошибки в журнал записывается информация с уровнем важности *Error*. Это не требует никаких изменений в приложении: когда возникает ошибка, возбуждается исключение — автоматически или специальным кодом. Инфраструктура автоматически регистрирует исключение в журнале. Это не относится к благополучно перехваченным и обработанным исключениям, впрочем, такие исключения не представляют настоящие ошибки — это нормальная последовательность событий в программе. Исключение перехватывается, а значит, ошибка обрабатывается.
- Информация о работе прикладных процессов записывается с уровнем важности *Info*. Одним из примеров является успешная регистрация

пользователя. Также должны регистрироваться наиболее важные крайние случаи — например, использование при регистрации недопустимого или уже зарегистрированного адреса электронной почты. По этим записям в файле журнала можно получить статистическую информацию о количестве регистраций. Кроме того, позднее можно проверить, пытался ли зарегистрироваться некоторый клиент и почему попытка оказалась безуспешной. Это позволит тщательно проверить ошибку и дать более развернутый ответ клиенту в случае его обращения в службу поддержки.

- В код также можно интегрировать запись в журнал сообщений с уровнем *Debug*. Однако это имеет смысл, только когда выполняется поиск конкретной ошибки. Поскольку в примере приложения этого не требуется, его код не выводит такие сообщения.

С помощью этой информации анализируются причины ошибок в приложении. Кроме того, файлы журналов дают возможность оценить, какие прикладные процедуры выполняются и как часто. Существуют некоторые решения для анализа файлов журналов, дающие возможность автоматически извлекать пары ключ/значение. Например, если в журнале появляется строка `firstname="Eberhard"`, ее можно отыскать и интерпретировать. Включение в журнальные записи имени переменной `firstname` с ее значением упрощает выделение всех записей, имеющих отношение к конкретному пользователю.

8.4. Анализ журналов примера приложения

Файлы журналов, генерируемые примером приложения, также можно анализировать. В данном случае это реализуется системой контейнеров Docker, в каждом из которых установлена часть инфраструктуры.

На рис. 8.2 показана фактическая организация этой системы.

- Приложение установлено в контейнере `user-registration`. Веб-интерфейс доступен через порт 8080, который отображается в порт 8080 виртуальной машины Vagrant и порт 8080 хоста. То есть приложение доступно по URL `http://localhost:8080/`. Файл журнала приложения находится в каталоге `/log`. Этот каталог отображается в том `/log`.
- Этот том монтируется также контейнером `logstash`. Этот контейнер осуществляет парсинг файлов из этого тома и сохраняет результаты

в Elasticsearch. Для этого в контейнере `logstash` используется соединение с контейнером `elasticsearch`, что обеспечивает взаимодействие двух контейнеров через общий порт.

- Контейнер `kibana` включает веб-сервер с приложением Node.js для доставки HTML и JavaScript браузеру. Этот веб-сервер выполняется в контейнере и доступен через порт 5601, который отображается в порт виртуальной машины Vagrant VM и хоста с тем же номером. Доступ к данным, хранящимся в Elasticsearch, осуществляется через соединение между контейнерами.

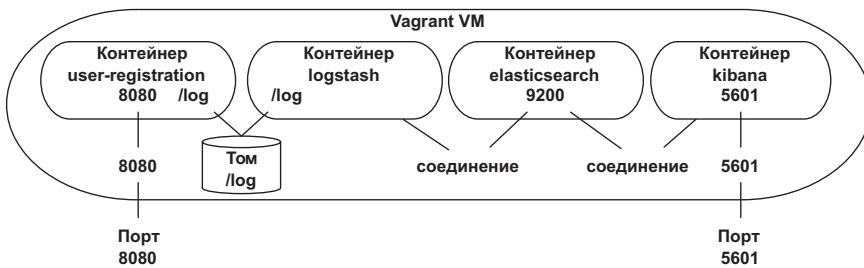


Рис. 8.2. Организация системы обработки журналов в примере приложения

Отдельные контейнеры Docker в этой системе используются как компоненты, взаимодействующие друг с другом через сетевые соединения или общую файловую систему, как описывалось в главе 3, «Автоматизация сборки и непрерывная интеграция».

Листинг 8.1. Конфигурация Logstash

```

input {
  file {
    path => ["/log/spring.log"]
    start_position => beginning
  }
}

filter {
  multiline {
    pattern => "((^\\s*)[a-z$\\.A-Z\\.]*Exception.+)|((^\\s*)at .+)"
    what => "previous"
  }
}

grok {
  match => [ "message",
    "^(?<date>[0-9]{4}\\-[0-9]{2}\\-[0-9]{2})"
  ]
}

```

```

    %{TIME:time}
    (?:\s*)
    (?<level>[A-Z]+)
    %{NUMBER:pid}
    (?:\-\-\-)
    (?<thread>\[.*\])
    (?<class>[0-9a-z$A-Z\[\/\.]*)
    (?:\s*:\s)
    (?<logmessage>(.\s)+)"
  }
  kv {}
}

output {
  elasticsearch {
    hosts => ["elasticsearch"]
  }
}

```

Особенный интерес в этой системе представляет конфигурация Logstash (листинг 8.1). Logstash использует конвейер: поступающие данные читаются с входа, фильтруются и затем записываются в вывод. Как определено в разделе `input`, Logstash читает данные с самого начала из файла журнала, путь к которому определяется параметром `path`. Параметр `multiline` в разделе `filter` требует, чтобы записи в файле журнала, располагающиеся в нескольких строках, объединялись в одно сообщение. Это необходимо для обработки исключений Java: они содержат информацию о месте, где возникла ошибка, располагающуюся в нескольких строках. С этой целью определяется регулярное выражение, совпадающее со строками, которые начинаются с имени исключения Java или со строки «at». Фильтр `grok` разбивает сообщения в поле `message` на отдельные поля в соответствии с регулярным выражением.

1988-10-12	19:42:07.350	INFO	683	---	[http-nio-8080-exec-6]	c.e.u.service.RegistrationService	email=eberhard.wolff@gmail.com deleted
date	time	level	pid		thread	class	logmessage

Рис. 8.3. Пример обработки записи в файле журнала

На рис. 8.3 показано, как именно обрабатывается запись в файле журнала: сначала сохраняется дата в поле `date` и время в поле `time`. Затем в поле `level` сохраняется уровень важности записи, в поле `pid` — идентификатор процесса, в поле `thread` — имя текущего потока выполнения и в поле

`class` — имя класса Java, сгенерировавшего запись в журнале. Наконец, в поле `logmessage` сохраняется текст самого сообщения.

Имя последнего фильтра `kv` — это сокращение от `key/value` (ключ/значение). Если запись в журнале содержит элементы, такие как `email=eberhard.wolff@gmail.com` (см. рис. 8.3), создается поле с именем `email` и значением `eberhard.wolff@gmail.com`. Это здорово упрощает поиск записей в журнале, соответствующих определенному клиенту.

Наконец, раздел `output` в конфигурации Logstash определяет, что данные экспортируются в Elasticsearch, в частности в индексы `log`. На основе выделенных полей Logstash генерирует JSON-документ, потому что этот формат поддерживается инструментом Elasticsearch. Elasticsearch может выполнять поиск любых значений в любых полях, а также генерировать статистики.

Сервер Elasticsearch определяется именем хоста `elasticsearch`. Настройки соединений в Docker гарантируют, что контейнер Docker с экземпляром Elasticsearch будет доступен по этому имени.

8.4.1. Анализ с применением Kibana

С помощью Kibana пользователи могут анализировать данные, собранные в Elasticsearch, непосредственно из браузера. Для этого в пользовательском интерфейсе Kibana имеется несколько вкладок. Начинать анализ предпочтительнее с вкладки `Discover` (Анализ). Здесь отображаются все поля, поэтому пользователи смогут рассмотреть все сообщения и интерпретировать их. Дополнительно на этой вкладке можно определить распределение значений, ограничить вывод отдельными полями или отфильтровать вывод так, чтобы отобразить только записи с определенными значениями. На рис. 8.4 изображен скриншот, который демонстрирует распределение записей по разным уровням важности.

Кроме того, существует возможность проанализировать данные в графическом виде — например, распределение зарегистрированных адресов электронной почты по доменам (рис. 8.5). Инструменты Elasticsearch и Kibana могут намного больше, чем просто искать данные, — например, они способны собирать статистическую информацию. Elasticsearch оптимизирован для максимальной скорости отклика, поэтому анализ выполняется очень быстро.

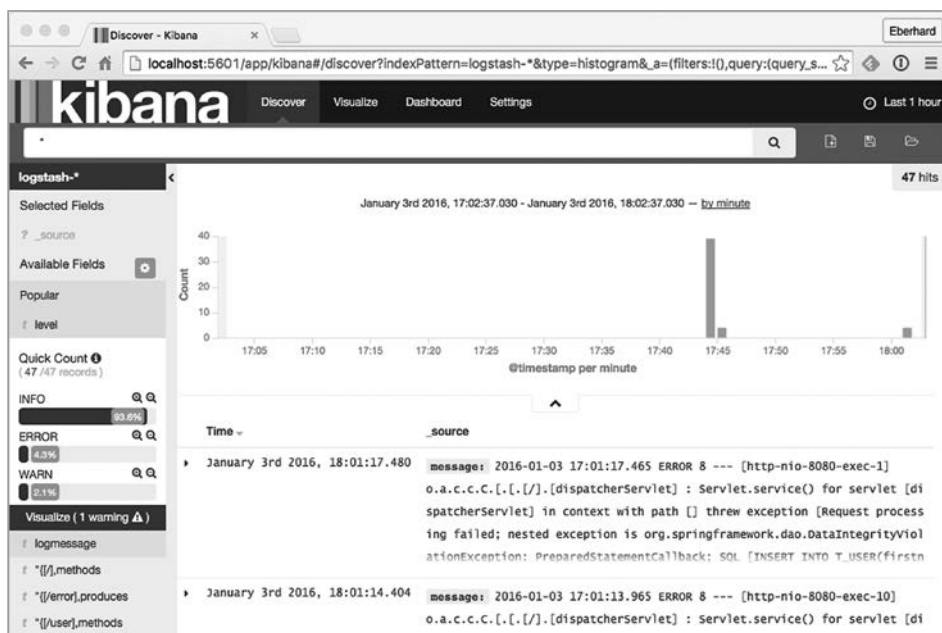


Рис. 8.4. Анализ данных с помощью Kibana

8.4.2. ELK — масштабируемость

Объем накапливаемых данных для примера приложения невелик и легко поддается управлению. Тем не менее масштабируемость является одной из самых сильных сторон стека ELK.

- Elasticsearch может делить индексы на сегменты (shard). Каждая запись помещается в сегмент. Поскольку сегменты могут храниться на разных серверах, такое деление позволяет равномерно распределить нагрузку. Сегменты могут также копироваться на несколько серверов для большей устойчивости к сбоям. Кроме того, операции чтения могут быть нацелены на произвольные копии (реплики) данных. Соответственно, операции чтения также могут масштабироваться с применением реплик.
- В отсутствие дополнительных настроек Logstash записывает данные за каждый день в отдельный индекс. Так как чаще всего читаются данные за текущую дату, такое деление помогает уменьшить объем данных, которые нужно просмотреть при выполнении типичных запросов, и тем самым увеличить производительность. Существуют также другие способы распределения данных по индексам — например, по пользователям.

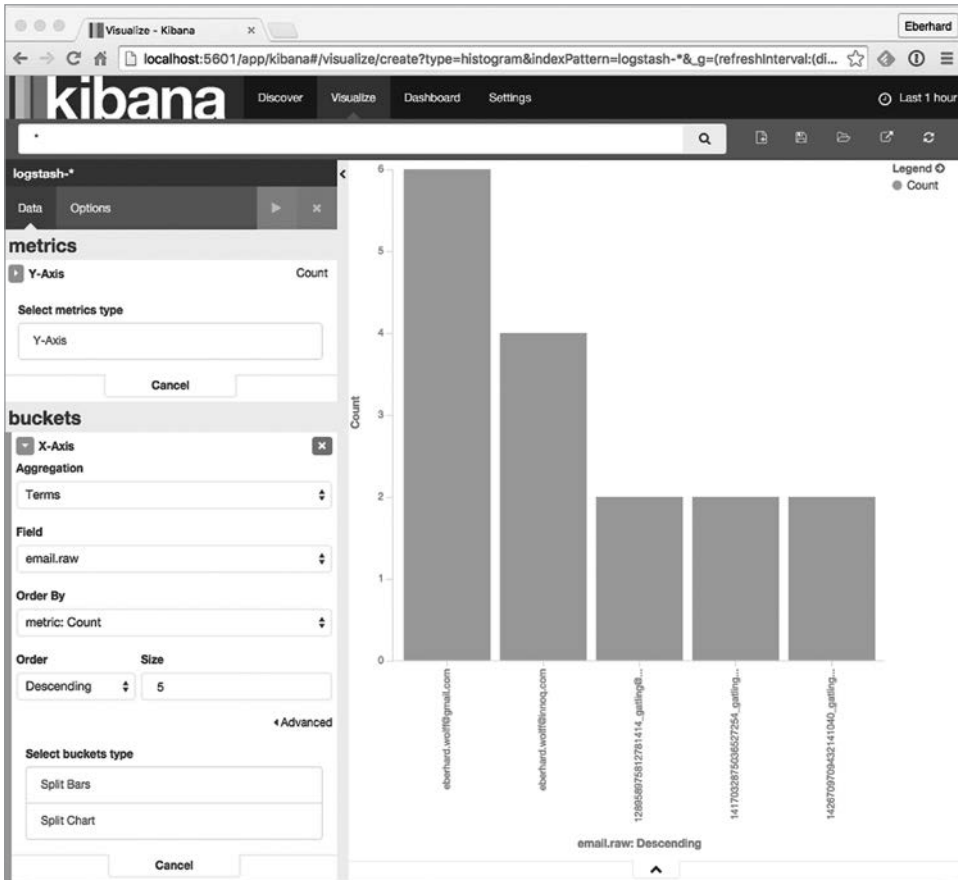


Рис. 8.5. Графический анализ в Kibana

- Согласно конфигурации, представленной выше, Logstash должен выполнять парсинг каждой строки в файле журнала. Было бы эффективнее, если бы приложение посылало информацию в журнал в виде документов JSON, потому что в этом случае можно полностью исключить этап парсинга и тем самым существенно уменьшить нагрузку на Logstash при наличии больших объемов данных.
- Наконец, Logstash поддерживает возможность масштабирования. Согласно конфигурации, приведенной выше, Logstash решает несколько задач: сбор данных с сервера и их парсинг. Для масштабирования эти задачи можно разделить и выполнять с применением разных технологий: читать данные с помощью инструмента транспортировки, бу-

феризовать с помощью брокера и обрабатывать с помощью Logstash. Это дает возможность установить Logstash на другом сервере, отдельно от приложения, и избавить центральный процессор сервера от интенсивных вычислительных операций, связанных с фильтрацией и обработкой данных в файле журнала. Брокер служит буфером, если накапливается слишком много записей, которые не могут быть обработаны немедленно. В качестве брокера часто используется Redis [6] — быстрая база данных в оперативной памяти. Роль инструмента транспортировки может играть сам Logstash. В этом случае он просто читает локальные файлы журналов и передает информацию брокеру. Это возможно, потому что Logstash поддерживает работу с самыми разными источниками и потребителями данных. В книге «Logstash Book» [7] подробно обсуждается, как настраивать и масштабировать Logstash.

- В числе других интересных инструментов можно назвать Filebeat [8], Beaver [9] и Woodchuck [10] — специализированные инструменты транспортировки. Обычно Filebeat выглядит предпочтительнее, потому что поддерживается компанией Elastic, ведущей разработку стека ELK. Кроме того, можно использовать стандартные инструменты UNIX, такие как syslog.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Скопируйте репозиторий с проектом примера (<https://github.com/ewolff/user-registration-V2>) с помощью инструмента командной строки (<http://git-scm.com/>): `git clone https://github.com/ewolff/user-registration-V2.git`.

- ▲ Окружение для анализа файлов журналов со стеком ELK находится в подкаталоге `log-analysis`.
- ▲ Установите Vagrant (<http://www.vagrantup.com/downloads.html>).
- ▲ Запустите окружение командой `vagrant up`.
- ▲ После этого вам будут доступны следующие URL:
 - <http://localhost:8080/> — приложение;
 - <http://localhost:5601/> — Kibana.

- ▲ Выполните несколько операций с приложением. Как вариант, можно изменить нагрузочные и приемочные тесты так, чтобы они взаимодействовали с выполняющимся приложением, а не запускали свою копию. Затем используйте нагрузочный тест для создания нагрузки на приложение.
 - Подсказка: ввод слишком длинного имени (>30 символов) будет приводить к появлению ошибки в приложении.
- ▲ Откройте главную страницу Kibana по адресу *http://localhost:5601/*.
- ▲ Настройте индекс.
 - Примите значение по умолчанию `logstash-*` для параметра `Index name or pattern` (Имя индекса или шаблон).
 - В параметре `Time-field name` (Имя поля с временем) укажите значение `@timestamp`.
- ▲ Перейдите на вкладку `Discover` (Анализ).
 - В правом верхнем углу установите период времени для отображаемых сообщений из журнала. По умолчанию отображаются данные за последние 15 минут — измените это значение, если необходимо.
 - Слева можно выбрать поле. После этого будут отображены все значения в этом поле.
- ▲ Щелкните на значке с изображением лупы рядом с одним из значений, чтобы активировать фильтр.
- ▲ Попробуйте отыскать конкретные записи. Для этого достаточно ввести искомую фразу в поле ввода в верхней части страницы.
- ▲ Попробуйте отыскать ошибки (записи с уровнем *Error*), но прежде вам нужно создать такие записи (например, попробовав указывать слишком длинные имена, >30 символов).

Описание других способов анализа можно найти по адресу: <https://www.elastic.co/guide/en/kibana/4.3/getting-started.html>. Например, на вкладке `Visualize` (Визуализация) создается свой вариант графического анализа, а на вкладке `Dashboard` (Панель мониторинга) — своя панель мониторинга.

- ▲ В настоящий момент система записывает данные в файл журнала, а Logstash выполняет парсинг и передает результаты в Elasticsearch. Чтобы облегчить работу для Logstash, данные можно поставлять непосредственно в формате JSON.
- ▲ Информация, необходимая для изменения конфигурации журналирования, находится по адресу: [https:// docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-logging.html).
- ▲ В каталоге приложения `src/main/resource` требуется создать файл `logback-spring.xml`. За дополнительной информацией обращайтесь по адресу: <https://github.com/spring-projects/spring-boot/tree/master/spring-boot-samples/spring-boot-sample-logback/>.
- ▲ После этого можно использовать кодер JSON; подробности ищите на странице: <https://github.com/logstash/logstash-logback-encoder>. Данные все еще необходимо записывать в файл; поэтому требуется только добавить запись для `LogstashAccessEncoder`, как описывается в документации, по адресу: <https://github.com/logstash/logstash-logback-encoder#encoder>.
- ▲ Наконец, измените конфигурацию Logstash и отключите парсинг данных.
- ▲ Данные в журнале необходимы только для анализа с применением стека ELK. Поэтому нет большого смысла записывать их в файл. Измените конфигурацию так, чтобы данные передавались напрямую, через сеть. Для этого расширьте конфигурацию JSON, настроив Logstash на прием информации непосредственно из сети, по протоколу TCP; подробности можно найти в документации, по адресу: <https://github.com/logstash/logstash-logback-encoder#tcp>.
- ▲ Измените конфигурацию так, чтобы информация от приложения передавалась инструментом Logstash непосредственно брокеру Redis, а затем фильтровалась и анализировалась. Это трудоемкая задача и требует, помимо всего прочего, установки дополнительных контейнеров Docker.
- ▲ Создайте новый образ и новый контейнер с брокером Redis. Подсказка: найдите пакет для Ubuntu с названием `redis-server`.
- ▲ Измените настройки Logstash так, чтобы он просто читал данные из файла журнала и передавал их брокеру Redis.

- ▲ Сгенерируйте новый экземпляр Logstash, который будет читать данные из Redis, выполнять их парсинг и записывать результаты в Elasticsearch. Для этого потребуется создать новый контейнер Docker.
- ▲ Используйте Docker Compose вместо Vagrant для запуска окружения ELK. Подходящую конфигурацию можно найти в файле `docker-compose.yml`. Описание использования Docker Compose можно найти в разделе 2.5.7.

8.5. Другие технологии обработки журналов

Разумеется, стек ELK — не единственный способ анализа файлов журналов. Существует также Graylog [11] — решение с открытым исходным кодом, тоже использующее Elasticsearch для сохранения файлов журналов. Кроме того, для хранения метаданных оно использует MongoDB. Однако Graylog определяет свой формат сообщений в журнале: GELF (Graylog Extended Log Format — расширенный формат журнала для Graylog), который также стандартизован для передачи данных. Расширения для поддержки GELF имеются во многих библиотеках журналирования и языках программирования. Кроме того, соответствующую информацию можно извлекать из журналов разного вида или собирать с помощью инструмента syslog в ОС Unix. Logstash также поддерживает GELF как формат входных и выходных данных. Graylog2 имеет веб-интерфейс, позволяющий осуществлять анализ информации из журнала.

Splunk [12] — коммерческое решение, давно присутствующее на рынке. Может расширяться большим количеством плагинов. Существуют также готовые решения для определенных инфраструктур, таких как Microsoft Windows Server. Программное обеспечение необязательно устанавливать в своем вычислительном центре; оно доступно также в виде облачной услуги. Решение Splunk используется для анализа не только файлов журналов, но и любых машинных данных. Для этой цели существует версия Splunk, которая называется Splunk Hunk и может интегрироваться с платформой Hadoop Big Data. В процессе обработки журналов Splunk сохраняет и индексирует данные для последующего поиска. Ориентация продукта на корпоративный сектор подчеркивается особым вниманием к безопасности. Поддерживается возможность анализа и вывода предупреждений в случае появления определенных проблем.

Чтобы избежать установки инфраструктуры для анализа файлов журналов, воспользуйтесь облачным решением. Такие решения предлагают готовые средства анализа и хранения данных. В этом случае ни установка, ни масштабирование не являются проблемой. Однако файлы журналов необходимо передать в облако, а для этого, конечно же, требуется подключение к Интернету с большой пропускной способностью. Кроме того, подобные решения просто непригодны в некоторых ситуациях — например, из-за ограничений по безопасности данных.

Одно из таких решений — Splunk Cloud — уже было представлено выше. В числе других альтернатив можно перечислить следующие.

- Loggly [13] — не самое сложное решение, позволяющее анализировать файлы журналов без особых усилий.
- Sumo Logic [14] — обладает намного более широкими возможностями. Кроме всего прочего, позволяет распознавать аномалии. Обладая средствами поддержки безопасности, в большей степени подходит для использования в корпоративном секторе.
- Papertrail [15] — очень простое решение, предназначенное в основном для объединения и анализа нескольких файлов журналов. Имеет текстовый пользовательский интерфейс. Papertrail не обладает большими возможностями, но поиск, конечно же, поддерживается.

Большинство предложений можно опробовать бесплатно, так что путь для экспериментов открыт. Основой для таких экспериментов может послужить пример приложения.

Некоторые решения для мониторинга, упомянутые в разделе 8.10, также способны обрабатывать файлы журналов. Это особенно актуально для коммерческих решений.

8.6. Продвинутое технологии обработки журналов

В области журналирования доступно большое количество инструментов и решений. Для записи в файл журнала в примере приложения используется Java-библиотека. Типичный подход к решению этой задачи, по крайней мере в Linux, заключается в использовании службы, такой как syslog-ng [16], реализующей централизованную систему журналирования, которая

может принимать информацию из разных источников и записывать ее в файлы журналов. Это позволяет использовать общую инфраструктуру журналирования для всех приложений.

8.6.1. Анонимизация

Еще одна проблема, возникающая при журналировании, особенно в рабочих окружениях, — защита данных. Данные, генерируемые в рабочем окружении, часто содержат личные данные пользователей, которые должны защищаться от несанкционированного доступа и, соответственно, не могут просто так передаваться разработчикам. Такие данные должны обезличиваться (анонимизироваться) перед передачей для анализа. Для этого в Logstash имеется фильтр с названием `anonymize`, который использует хэши. Хэши гарантируют однозначность идентификации — то есть значение хэша всегда однозначно соответствует имени пользователя, появляющемуся в файле журнала. Однако на основе хэша существует возможность определить имя пользователя.

8.6.2. Производительность

В большинстве систем используется несколько серверов и разные приложения, реализующие необходимые функциональные возможности. Для анализа производительности таких систем требуется изучить последовательность действий, выполняемых разными серверами. Для этого в каждом запросе следует использовать однозначную строку, которая включается в каждое журналируемое сообщение. Если сообщения из журнала собираются на центральном сервере и там же выполняется поиск по всем запросам с тем же значением, это дает возможность проверить, какие компоненты системы использовались и сколько времени было затрачено на обработку запросов. Существуют специализированные решения, такие как Zipkin [17], поддерживающие не только сбор данных, но также извлечение и отображение результатов.

8.6.3. Время

В ходе журналирования часто возникает проблема рассинхронизации часов на разных компьютерах. Это осложняет точный анализ взаимозависимостей. Logstash снабжает каждую запись отметкой времени. Поэтому про-

блему рассинхронизации можно обойти, если синхронизировать часы хотя бы на компьютерах, где действует Logstash.

8.6.4. Эксплуатационная база данных

Постепенно в централизованном хранилище, таком как Elasticsearch, будет накапливаться все больше и больше данных о параметрах настройки, производительности (времени начала и окончания) и статистика о работе процессов. Аналогично можно собирать и сохранять данные мониторинга. В результате получится база данных, содержащая полную информацию о работе приложения. Информация из этой базы данных может использоваться для планирования расширения производственных мощностей или поиска направлений для оптимизации — например, определения областей, где ошибки случаются чаще всего, или идентификации узких мест, в смысле производительности. То есть такая база данных является важным источником обратной связи, на которую опирается методология непрерывного развертывания для оптимизации программного обеспечения.

8.7. Мониторинг

Помимо анализа файлов журналов большой интерес представляют числовые характеристики функционирования приложения. Их измерение помогает пролить свет на текущее состояние приложения и на то, как оно меняется с течением времени. Это дает возможность заранее предсказывать проблемы, такие как исчерпание места на жестком диске. Благодаря постоянному мониторингу можно своевременно предупредить о проблеме и предотвратить ее до того, как она достигнет критического уровня, — например, удалив ненужные файлы. То есть числовые характеристики функционирования (или метрики) являются важной частью мониторинга системы и приложений. Любая группа оперативного сопровождения сможет установить свое решение, охватывающее эту область.

Мониторинг должен осуществляться с помощью внешнего процесса. Только внешний процесс может определить, запущен ли процесс приложения, и послать сигнал тревоги, если это не так. Кроме того, это единственный способ гарантировать доступность данных мониторинга в периоды, когда прикладной процесс работает под высокой нагрузкой. В таких ситуациях

функции мониторинга в самом прикладном процессе могли бы перестать откликаться или откликаться с большой задержкой.

В контексте непрерывного развертывания мониторинг и его результаты так же важны, как анализ файлов журналов. Они позволяют судить о работе приложения и принимать решения о направлениях дальнейшего его развития. Например, технологии мониторинга позволяют собирать не только техническую информацию, но также сведения о доходах от продаж и связывать их с техническими характеристиками. Когда объем продаж внезапно падает, это может быть связано с проблемами в приложении. Такие зависимости часто достаточно очевидны. Поэтому внимание должно уделяться не только техническим характеристикам, но и данным, имеющим большое значение для бизнеса. В конце концов, целью программного обеспечения является поддержка бизнес-процессов. Измерение ценности приложения для бизнеса выглядит вполне логично. Если, например, система функционирует с технической точки зрения, но продажи отсутствуют, этот факт должен определяться при мониторинге и вызывать отправку сигнала тревоги, потому что это явный отказ системы с точки зрения бизнеса.

8.8. Отображение числовых характеристик с помощью Graphite

В большинстве окружений уже имеется система мониторинга, объединяющая все серверы и позволяющая информировать администраторов о проблемах, даже в нерабочие часы. Поэтому новые приложения часто просто интегрируются в существующую систему мониторинга. В порядке обмена опытом, чтобы вы могли получить представление о возможностях инструментов мониторинга, в этом разделе мы познакомимся с Graphite [18]. Graphite — это инструмент для обработки числовых характеристик, представляющий законченное решение.

- Graphite хранит только числовые данные. Это ограничение вполне приемлемо: типичные характеристики, определяемые в ходе мониторинга, такие как потребление ресурсов или время отклика, легко могут быть представлены в числовой форме.
- Graphite специализирован для обработки динамических последовательностей, потому что прикладные данные часто представляют интерес только с точки зрения их изменения с течением времени.

- Данные отображаются веб-приложением, благодаря чему пользователи относительно легко могут анализировать их.

Graphite включает три компонента.

- *Carbon* принимает данные и сохраняет их во внутреннем кэше.
- *Whisper* — простая библиотека для обработки динамических последовательностей данных. Этот компонент хранит данные только за определенный интервал времени, потому что какое-то время спустя результаты мониторинга теряют свою актуальность.
- Наконец, *веб-приложение Graphite* открывает доступ к собранным данным.

На рис. 8.6 изображен пример пользовательского интерфейса веб-приложения Graphite. Помимо времени отклика отображается также количество запросов к определенным адресам URL, обслуживаемым примером прило-

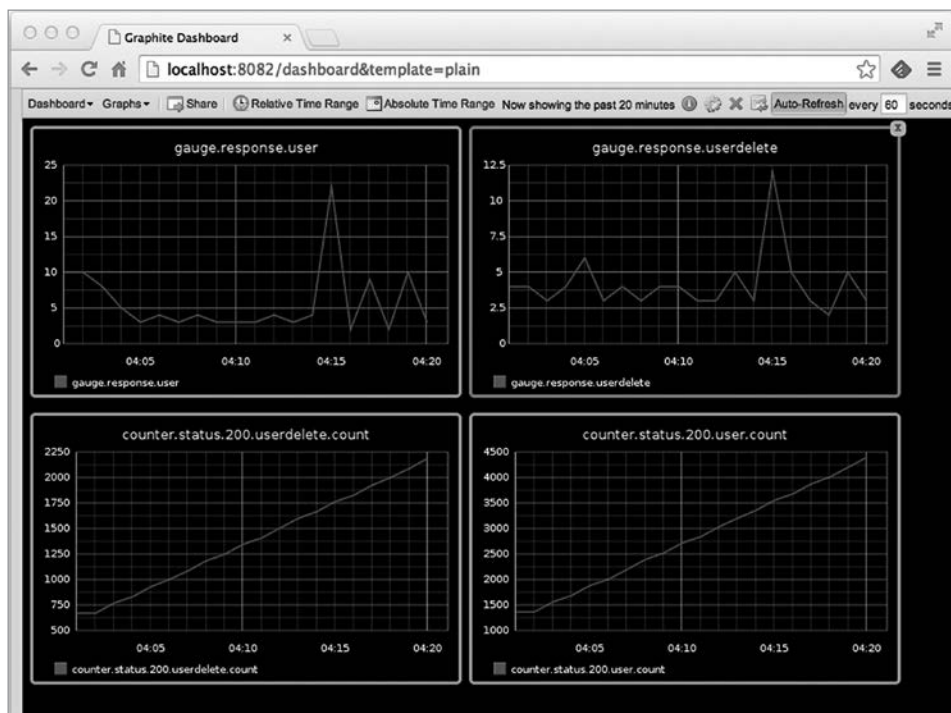


Рис. 8.6. Панель мониторинга Graphite

жения. Благодаря этому можно не только следить за работой приложения, но также оценивать его поведение.

Панель различает два вида данных.

- *Датчики (gauges)* — числовые значения, изменяющиеся с течением времени, для которых можно определить среднее значение. Графики на рис. 8.6, отражающие изменение времени отклика, как раз соответствуют датчикам.
- *Счетчики (counters)* могут увеличиваться или уменьшаться. В панели они представляют количество обращений к адресам URL, связанным с функциями удаления учетных записей и получения списка пользователей.

Значения для передачи инструменту Graphite имеют очень простую структуру: собственно значение, имя характеристики (метрики) и отметка времени.

В рабочем окружении Graphite может дополняться другими инструментами.

- StatsD [19] собирает значения и передает их в Graphite. Это позволяет уменьшить объем данных для передачи в Graphite и тем самым уменьшить нагрузку на него.
- Grafana [20] добавляет в Graphite поддержку альтернативных панелей и других графических элементов.
- Collectd [21] собирает статистическую информацию о системе, например потребление времени центрального процессора. Эти данные можно анализировать с применением специализированных интерфейсов или хранить в Graphite.
- Seyren [22] добавляет в Graphite возможность отправки сигналов тревоги.

Существует большое количество решений для мониторинга. Список, приведенный выше, дает лишь первое впечатление. Проведение мониторинга также целесообразно с точки зрения бизнеса. Например, когда предполагается методом А/В-тестирования определить лучшую из двух альтернатив, требуется сначала измерить и сохранить определяющие характеристики.

8.9. Характеристики, измеряемые в примере приложения

В примере приложения измеряемые характеристики записываются с помощью Metrics [23]. Это специализированная Java-библиотека, предназначенная для записи метрик. В первую очередь с ее помощью записывается информация о HTTP-запросах. Это позволяет определить частоту обращений к определенному URL и интервалы времени, потребовавшиеся для отправки ответов. Аналогично записываются собственные метрики. Библиотека Metrics имеет расширение для поддержки Graphite, позволяющее посылать значения непосредственно в Graphite для дальнейшей оценки пользователем.

8.9.1. Структура примера

В примере Graphite устанавливается в три контейнера Docker. В процессе установки некоторые файлы в контейнере Docker перезаписываются. На рис. 8.7 изображена общая схема системы мониторинга. Пример приложения выполняется в собственном контейнере. Доступ к приложению осуществляется через порт 8083.

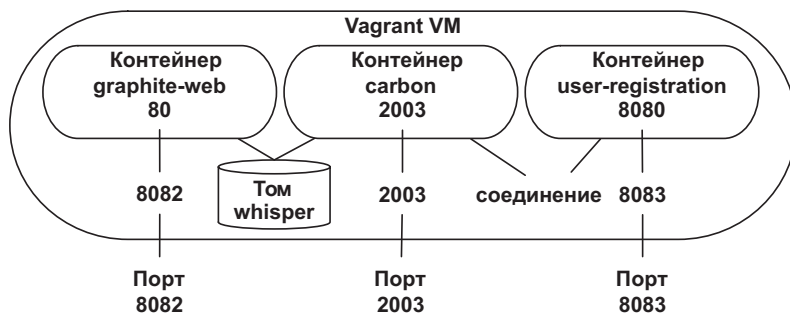


Рис. 8.7. Схема установки Graphite

Carbon принимает данные мониторинга и выполняется в другом контейнере. Передача данных в Carbon осуществляется через порт 2003. Этот порт доступен извне через соединение между контейнерами приложения. Благодаря этому хост может передавать данные в Graphite. То есть такую установку Graphite легко можно использовать в других контекстах. Carbon

также поддерживает масштабирование — хранилище данных может быть распределено между несколькими серверами для поддержки больших объемов данных. Для внутреннего хранения данных Carbon использует Whisper и записывает их в выделенный том.

Веб-приложение Graphite доступно хосту по адресу *http://localhost:8082/*. Оно читает данные из тома, куда они записываются контейнером Carbon, и отображает их для анализа.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Выберите проект, который вы хорошо знаете.

- ▲ Какие инструменты используются в нем для мониторинга?
- ▲ Какая информация передается приложением в систему мониторинга?
- ▲ Эта информация имеет исключительно технический характер или бизнес-характеристики тоже записываются?
- ▲ Позволяет ли система мониторинга идентифицировать проблему, если важные для бизнеса функции, такие как прием оплаты, терпят неудачу? Например, если она заметит отсутствие роста продаж?
- ▲ Извлеките проект примера из репозитория (*https://github.com/ewolff/user-registration-V2*). Используйте для этого инструмент командной строки Git (*http://git-scm.com/*) и выполните команду `git clone https://github.com/ewolff/user-registration-V2.git`.
- ▲ Окружение для мониторинга с Graphite можно найти в подкаталоге `graphite`.
- ▲ Установите Vagrant (*http://www.vagrantup.com/downloads.html*).
- ▲ Запустите окружение командой `vagrant up`.
- ▲ После этого станут доступны следующие URL:
 - *http://localhost:8083/* — приложение;
 - *http://localhost:8082/* — веб-приложение Graphite.
- ▲ Выполните несколько операций с приложением. Как вариант, можно изменить нагрузочные и приемочные тесты так, чтобы они взаи-

модействовали с выполняющимся приложением, а не запускали свою копию. Затем используйте нагрузочный тест для создания нагрузки на приложение.

- ▲ Создайте панель в интерфейсе Graphite, подобную изображенной на рис. 8.6.
- ▲ Загляните в конфигурацию Docker и определите, как реализованы соединения между контейнерами и доступ к портам.
- ▲ После установки Graphite в соответствующем контейнере требуется изменить некоторые файлы. Как это реализовано? Какие преимущества и недостатки имеет используемый подход в сравнении с применением шаблонов Chef?
- ▲ Дополните систему еще одним компонентом, например StatsD [24].
- ▲ Какую дополнительную информацию можно прочитать из проекта примера? Действительно ли количество регистраций является ценной информацией?
- ▲ Измените код так, чтобы счетчик регистраций сохранялся в Graphite. Так как приложение использует Spring Boot, это не потребует больших усилий.
- ▲ Используйте Docker Compose вместо Vagrant для запуска окружения с Graphite. Подходящую конфигурацию можно найти в файле `docker-compose.yml`. Описание использования Docker Compose можно найти в разделе 2.5.7.

8.10. Другие решения для мониторинга

Graphite — не единственное решение для мониторинга приложений и систем.

- Nagios [25] — исчерпывающее решение для мониторинга и может служить альтернативой Graphite.
- Incinga [26] — первоначально создавалось на основе проекта Nagios и затем было переработано. Однако все еще охватывает очень похожий контекст.

- Существуют также другие коммерческие решения, такие как HP Operations Manager [27], IBM Tivoli [28], CA Opscenter [29] и BMC Remedy [30]. Эти инструменты обладают самыми широкими возможностями, давно присутствуют на рынке и предлагают поддержку многочисленных программных и аппаратных продуктов. Такие платформы часто внедряются на уровне предприятия — и такие внедрения в действительности являются очень сложными проектами. Приложения должны интегрироваться в решение, чтобы соответствовать стандартам предприятия. Некоторые из этих решений способны анализировать и осуществлять мониторинг файлов журналов.
- Riemann [31] — инструмент с открытым исходным кодом. Принцип его действия основан на обработке потоков событий. Для этого в нем используется логика функционального программирования, определяющая реакцию на конкретные события. Существует возможность отображения результатов мониторинга в настроенной для этого панели или рассылки сообщений по электронной почте или посредством SMS.
- Функцию мониторинга можно также передать в облако, благодаря чему отпадает необходимость устанавливать и настраивать обширную инфраструктуру. Это упрощает ввод в эксплуатацию инструментов для мониторинга приложений. Примером может служить NewRelic [32].
- стек TICK [33] — законченный стек решений с открытым исходным кодом для мониторинга, основанный на InfluxDB. Это — база данных для служб времени, подобная Whisper, специализированная для хранения временных последовательностей данных и способная интегрироваться, например, с Grafana или Graphite. Также в состав стека входят другие инструменты: Telegraf — для сбора данных, Chronograf — для визуализации и Kapacitor — для автоматизированного поиска аномалий и рассылки предупреждений.
- Packetbeat [34] — использует инструменты Elasticsearch и Kibana из стека ELK для хранения и анализа данных. Это позволяет относительно легко комбинировать его со стеком ELK. Packetbeat осуществляет мониторинг сетевого трафика, но также способен исследовать содержимое сетевых пакетов и поставлять информацию, например, о конкретных запросах SQL. Этот подход особенно удобен для мониторинга распределенных приложений, поскольку упрощает получение и анализ информации обо всей системе.

8.11. Прочие проблемы, возникающие во время эксплуатации приложений

Во время эксплуатации приложения интерес представляют не только данные из журналов и средств мониторинга, а также учет изменений. Для непрерывного развертывания это важная проблема, потому что изменения в рабочем окружении в идеале должны производиться только путем изменения сценариев. Такие изменения легко отслеживаются с применением систем управления версиями — всегда можно выяснить, кто, когда и какое изменение произвел. То есть учет изменений в рабочем окружении без труда можно организовать с помощью системы управления версиями.

В этом случае отпадает необходимость иметь прямой доступ к серверам: информацию из рабочего окружения можно получать через файлы журналов и метрики. Развертывание новых версий может осуществляться автоматически. Это дает разные преимущества: например, улучшенная защищенность, потому что никто — включая хакеров — не сможет войти в систему. Помимо оперативного персонала разработчики также могут читать информацию из рабочего окружения, например, посредством системы, подобной стеку ELK. Это ускоряет анализ и исправление ошибок.

8.11.1. Сценарии

В процессе эксплуатации приложения иногда возникает необходимость изменить параметры настройки или как-то иначе взаимодействовать с ним. Такие операции должны автоматизироваться. Необходимость производить изменения вручную влечет увеличение расходов и вероятность появления ошибок. Автоматизированную процедуру можно повторять снова и снова без особых усилий — именно это является целью методологии непрерывного развертывания. Однако для этого необходимо иметь возможность автоматизировать подобные вмешательства с помощью сценариев. Наличие пользовательского интерфейса для администрирования приносит определенную пользу, но вдобавок к нему всегда хорошо иметь возможность вмешательства посредством интерфейса сценариев.

8.11.2. Приложения в вычислительном центре клиента

Подходы, представленные здесь, оптимизированы для случаев эксплуатации внутренних приложений. Однако нередко приложения устанавлива-

ются, например, в вычислительном центре клиента, как часть аппаратного комплекса, или на мобильные устройства. Это серьезно усложняет развертывание новых версий и мониторинг. Тем не менее принципы, изложенные ниже, также можно использовать в подобных случаях.

- Установка новых версий должна быть автоматизирована. В основном это относится к мобильным приложениям, потому что магазины приложений предлагают подобные возможности. Программное обеспечение, устанавливаемое в вычислительном центре клиента, с применением подобных механизмов поддерживается в актуальном состоянии. Обновления должны устанавливаться автоматически, благодаря этому приходится поддерживать меньшее количество версий, ошибки исправляются быстрее и проблемы с безопасностью решаются проще.
- Непрерывно получать файлы журналов от всех клиентов и приложений невозможно, в том числе и по соображениям безопасности. Тем не менее должна существовать возможность в случае ошибки послать все необходимые сведения по электронной почте или сделать их доступными другими средствами. Эта задача тоже должна быть автоматизирована, чтобы для ее выполнения требовалось только щелкнуть мышкой. Это гарантирует быстрое получение разработчиками необходимой информации.

В конечном итоге цель остается прежней: по возможности предоставить прямой доступ к необходимой информации и максимально упростить и автоматизировать развертывание нового программного обеспечения. В наши дни основная масса программного обеспечения распространяется через Интернет, и, соответственно, через него можно осуществлять доступ к информации из рабочих систем. Аналогичные подходы можно использовать для систем, которые устанавливаются у клиентов, в их вычислительных центрах.

8.12. В заключение

С технической точки зрения, при эксплуатации приложения основное внимание должно уделяться получению информации из рабочего окружения и ее анализу. В этой главе было показано два возможных решения этой задачи. Приложение выводит сведения в файлы журналов, которые затем анализируются с применением некоторой инфраструктуры, такой как стек ELK. Другим решением является мониторинг, в ходе которого в первую

очередь оцениваются числовые характеристики. В качестве примера этого решения был представлен инструмент Graphite. Благодаря этим решениям оперативный персонал может получать необходимую информацию из приложения. В то же время, обладая этими данными, разработчики и архитекторы получают возможность более целенаправленно продолжать разработку приложения. Без подобной обратной связи целенаправленная разработка едва ли была бы возможна.

Ссылки

1. <https://www.elastic.co/products/logstash>
2. <https://www.elastic.co/products/elasticsearch>
3. <https://www.elastic.co/products/kibana>
4. <http://commons.apache.org/proper/commons-logging/>
5. <http://logback.qos.ch/>
6. <https://redis.io/>
7. <https://www.logstashbook.com/>
8. <https://www.elastic.co/products/beats/filebeat>
9. <https://github.com/python-beaver/python-beaver>
10. <https://github.com/danryan/woodchuck>
11. <https://www.graylog.org/>
12. <https://www.splunk.com/>
13. <https://www.loggly.com/>
14. <https://www.sumologic.com/>
15. <https://papertrailapp.com/>
16. <https://www.balabit.com/network-security/syslog-ng>
17. <https://github.com/openzipkin/zipkin>
18. <http://graphiteapp.org/>
19. <https://github.com/etsy/statsd/>
20. <http://grafana.org/>

21. <https://collectd.org/>
22. <https://github.com/scobal/seiyren>
23. <http://metrics.codahale.com/> или <https://github.com/dropwizard/metrics>
24. <https://github.com/etsy/statsd/>
25. <https://www.nagios.org/>
26. <https://www.icinga.com/>
27. <http://www8.hp.com/us/en/software-solutions/operations-manager-infrastructure-monitoring/>
28. <https://www.ibm.com/software/tivoli>
29. <http://www3.ca.com/au/opscenter.aspx>
30. <http://www.bmc.com/it-solutions/remedy-itsm.html>
31. <http://riemann.io/>
32. <https://newrelic.com/>
33. <https://www.influxdata.com/>
34. <https://www.elastic.co/products/beats/packetbeat>

Часть III

УПРАВЛЕНИЕ, ОРГАНИЗАЦИЯ И АРХИТЕКТУРА РЕШЕНИЯ НЕПРЕРЫВНОГО РАЗВЕРТЫВАНИЯ

Методология непрерывного развертывания не просто технология, она имеет более широкую область применения. Об этом рассказывается в последней части книги.

- ✓ **Глава 9** показывает, как внедрить методологию непрерывного развертывания в своей организации.
- ✓ **Глава 10** демонстрирует связь между техническими и организационными приемами реализации методологии непрерывного развертывания (DevOps).
- ✓ Непрерывное развертывание также оказывает влияние на архитектуру приложений. Связанные с этим вопросы обсуждаются в **главе 11**.
- ✓ **Глава 12** представляет заключительные замечания и завершает эту книгу.

9

Внедрение методологии непрерывного развертывания на предприятии

9.1. Введение

Внедрение непрерывного развертывания по сути означает создание конвейера непрерывного развертывания. В разделе 9.2 показано, как это делается с самого начала нового проекта. Внедрение непрерывного развертывания в существующий проект в основном является задачей оптимизации, потому что конвейер передачи новых версий в рабочее окружение уже существует. Главный вопрос, который возникает в этом случае, — где и как начать оптимизацию конвейера. В разделе 9.3 описывается популярный подход к внедрению непрерывного развертывания — «Систематизация потока ценностей» (Value Stream Mapping), а в разделе 9.4 обсуждаются другие подходы, такие как «капиталовложения в качество», «останови конвейер» (Stop the Line) и «пять почему» (5 Whys).

9.2. Непрерывное развертывание с самого начала

Внедрение непрерывного развертывания осуществляется проще, если происходит с самого начала нового проекта. Когда реализация конвейера непрерывного развертывания начинается одновременно с проектом, существует возможность конструировать и развивать его поэтапно, вместе с проектом. Прежде всего следует начать работу над определенными этапами — например, выпуском (см. главу 3, «Автоматизация сборки и непрерывная интеграция») и развертыванием (глава 7, «Развертывание — ввод в эксплуатацию»), поскольку эти два этапа составляют минимальный набор, необходимый для передачи приложения в эксплуатацию. Впоследствии постепенно добавляются другие этапы — например, при-

емочные испытания и другие виды автоматизированного тестирования. В некоторых случаях в зависимости от нефункциональных требований можно обойтись без тестирования производительности и безопасности. Это позволяет осуществлять внедрение постепенно, с меньшими усилиями и затратами.

Кроме того, при внедрении непрерывного развертывания с начала разработки проекта существует возможность учитывать требования методологии при выборе технологий и архитектур. Если определенная база данных или технология для другого компонента архитектуры проще поддается автоматизации, ее можно рассматривать как более предпочтительную для использования. Сложность настройки приложения и подготовки инфраструктуры должна контролироваться с самого начала, поскольку иначе автоматизация будет труднодостижимой. Как следствие, с самого начала проводится оптимизация приложения с целью уменьшить сложность установки и эксплуатации.

Влияние выбора технологий на конвейер непрерывного развертывания имеет множество аспектов: например, выбор языка программирования влияет на скорость работы компилятора. Это, в свою очередь, влияет на время, затрачиваемое конвейером непрерывного развертывания. Решения, касающиеся выбора языка программирования, могут также влиять на качество конвейера непрерывного развертывания.

Наконец, приступить к реализации непрерывного развертывания с самого начала выгодно еще и потому, что конструирование и оптимизация конвейера происходят постепенно. Кроме того, это позволяет согласовывать выбор технологий и архитектуры с требованиями методологии непрерывного развертывания. Все это упрощает создание конвейера. В итоге получается, что внедрение методологии непрерывного развертывания осуществляется проще, если совпадает с началом разработки нового проекта.

9.3. Систематизация потока ценностей

Большинство проектов, построенных на основе существующего кода, разрабатывались без учета методологии непрерывного развертывания. Внедрение этой методологии в существующий проект лучше всего начинать с осмысления ее основ: она базируется на быстром получении обратной

связи и снижении затрат (см. раздел 1.4.10). Под снижением затрат здесь понимается уменьшение «отходов» — всего того, что не представляет ценности для заказчика. Примером таких «отходов» может служить код, не использующийся в рабочем окружении. Такой код может реализовать новые возможности, пока недоступные для заказчика. То есть целью непрерывного развертывания является не ожидание, когда большое количество изменений можно будет собрать воедино и развернуть, а быстрое развертывание небольших изменений. Новые функции и изменения в коде поступают в виде непрерывного потока, поэтому развертывание должно производиться регулярно. Это соответствует идеалам подхода Lean Startup, требующего постоянного протекания потока ценности через систему.

Конвейер непрерывного развертывания в том или ином виде существует всегда, потому что всегда существует комплекс процедур передачи программного обеспечения в эксплуатацию. Однако этот процесс может быть очень сложным или требовать много времени, из-за чего мало походит на непрерывный поток.

9.3.1. Систематизация потока ценностей описывает последовательность событий

Чтобы приблизиться к идеальному воплощению непрерывного развертывания с быстрой обратной связью, воспользуемся подходом систематизации потока ценности (Value Stream Mapping). Он описывает текущую последовательность событий вплоть до выпуска. Для каждого этапа определяется время обработки (Value Add Time) и время ожидания (Waste Time). Их сумма определяет общее требуемое время (Cycle Time). Эффективность процесса оценивается как отношение времени обработки к общему требуемому времени. То есть к идеалу можно приблизиться, уменьшив время ожидания и реализовав постоянное перетекание новых возможностей через конвейер. Должны ли учитываться затраты на устранение ошибок и других проблем в этих показателях? Безусловно, потому что исправление ошибок может занимать много времени. Поэтому на этапах тестирования требуется определить, что должно происходить при выявлении ошибок. Кроме того, в потоке ценностей могут быть определены контактные лица и необходимые ресурсы для оптимизации. При наличии узкого места, когда изменения недостаточно быстро преодолевают неко-

торый этап, между этапами возникает очередь. Если, например, скопилось много изменений, ожидающих ручного тестирования, это является признаком узкого места.

9.3.2. Оптимизация

Одной из возможностей оптимизации потока ценностей является внедрение ограничений для очередей. Это делает более очевидными узкие места — они совпадают с местами, где достигаются установленные ограничения. Данный прием помогает выявлять узкие места и ликвидировать их. Например, можно определить ограниченное количество изменений, ожидающих ручного тестирования. Когда очередь заполняется полностью, предыдущий этап может прекратить прием новых изменений — в самом крайнем случае это просто прервет реализацию новых функций. Зато вынудит разработчика приступить к ликвидации узкого места на этапе ручного тестирования.

Определение времени циклов также может способствовать выявлению потенциальных возможностей оптимизации. Когда выполнение какого-то этапа занимает слишком много времени или возникает длительное время ожидания, это явно указывает на проблемы в организации конвейера.

На рис. 9.1 показан простой пример. Выпуск, сборка и модульное тестирование, а также анализ кода выполняются на сервере непрерывной интеграции. Сервер выполняет эти этапы друг за другом. Время ожидания почти отсутствует — сервер непрерывной интеграции начинает работу практически сразу, как только разработчик сохранит изменения. Прохождение этапов также осуществляется относительно быстро. Однако приемочные испытания и тестирование пропускной способности

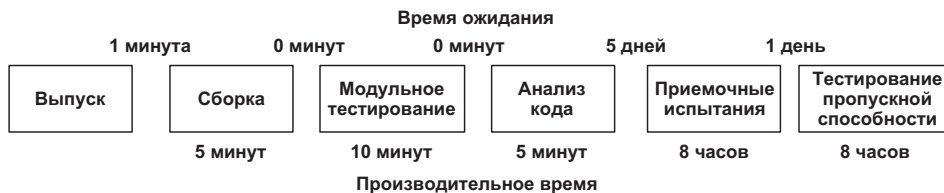


Рис. 9.1. Систематизация потока ценностей до оптимизации

продолжаются очень долго, из-за чего изменения вынуждены ждать некоторое время, прежде чем будут включены в эти этапы. Причина в том, что эти тесты выполняются вручную и только один раз, когда выпускается новая версия — например, только раз в две недели. Поэтому изменение вынуждено ждать в среднем пять дней, прежде чем сможет пройти эти тесты.

Систематизацию потока ценностей также имеет смысл применять, когда конвейер еще не автоматизирован полностью. С ее помощью выявляются потенциальные направления оптимизации и определяются необходимые мероприятия перед внедрением непрерывного развертывания.

Постепенно все узкие места в конвейере будут ликвидированы. На рис. 9.2 изображен тот же конвейер, но уже существенно автоматизированный после нескольких циклов оптимизации — в основном этапов приемочных испытаний и тестирования пропускной способности. Теперь они выполняются дополнительной системой, отдельной от сервера непрерывной интеграции. Однако эти тесты все еще имеют некоторое время ожидания. Чтобы уменьшить его, можно попробовать сократить время работы тестов. После сокращения времени обработки тестов сократится и время ожидания, потому что компьютеры быстрее освобождаются для выполнения следующих задач.



Рис. 9.2. Систематизация потока ценностей после оптимизации

Существуют различные подходы к оптимизации времени обработки приемочных тестов и тестов пропускной способности, например использование более производительного аппаратного обеспечения, усовершенствование тестов или создание нескольких окружений для тестирования. Кроме того, тестирование можно разделить так, чтобы сначала проводились более простые тесты, выполняющие тестирование «в ширину», и только потом более глубокие и всесторонние тесты, оценивающие код «в глубину». Это обеспечит более быструю обратную связь для разработчиков, когда какая-то функция в принципе окажется неработоспособной.

Как видите, прием систематизации потока ценностей может быть полезным инструментом для постепенной оптимизации существующего конвейера.

9.4. Дополнительные меры для оптимизации

Систематизация потока ценностей помогает выявить, где можно оптимизировать конвейер непрерывного развертывания. Однако в общем случае поиск этапов, подлежащих улучшению, не вызывает проблем — сложнее выбрать, что оптимизировать и в каком месте.

9.4.1. Капиталовложения в качество

Когда приходит момент принятия решения, может пригодиться метафора *капиталовложений в качество* [1]. Основная идея заключается в оценке затрат и выгод для всех предполагаемых мер и выборе тех из них, что обеспечивают лучшее соотношение затраты/выгоды. Затраты — это усилия, которые необходимо приложить для реализации оптимизации. Применительно к автоматизации тестов в затраты входит создание тестов и их интеграция в конвейер непрерывного развертывания.

9.4.2. Затраты

Оценка затрат особенно важна, когда методология непрерывного развертывания внедряется в существующий проект, потому что затраты на внедрение некоторых мер могут оказаться очень высокими, если какие-то характеристики программного обеспечения ранее не принимались во внимание. Часто разработчиков не интересует, насколько просто или сложно производится установка программных пакетов. Может так случиться, что разработчики реализуют очень гибкие возможности конфигурации, которые, однако, создают дополнительные затраты в ходе эксплуатации и в итоге приводят к существенным издержкам. Высокая сложность настройки может серьезно препятствовать автоматизации процесса установки. Методология непрерывного развертывания предполагает необходимость установки программного обеспечения на разных этапах для тестирования. Поэтому возможность автоматизированной установки действительно является очень желательной. Однако для ее достижения могут потребоваться настолько значительные усилия, что в конечном итоге предпочтительнее отказаться от полной автоматизации.

9.4.3. Выгоды

Затраты компенсируются выгодами. Внедрение методологии непрерывного развертывания дает две основные выгоды: снижение усилий и увеличение надежности. Например, автоматизация приемочных испытаний уменьшает трудозатраты на их проведение. В то же время надежность результатов тестирования увеличивается, потому что они могут точно воспроизводиться благодаря автоматизации.

Кроме этой прямой выгоды имеется также выгода косвенная: автоматизация позволяет проводить тестирование чаще. Это увеличивает вероятность обнаружения ошибок на самых ранних этапах и в итоге ведет к повышению качества. Когда тесты выполняются чаще, разработчики быстрее получают обратную связь. Это упрощает устранение ошибок, потому что разработчики не будут спешить с внесением изменений до выявления проблем в ходе тестирования. Кроме того, они будут помнить свои последние изменения более отчетливо, и, соответственно, им проще будет идентифицировать причину ошибки.

На первый взгляд определение затрат и выгод кажется простой задачей — в конце концов, достаточно лишь оценить расходы, связанные с реализацией мер, и получаемый прирост производительности/надежности. Однако в действительности детальный анализ осложнен наличием прямых и косвенных выгод, которые трудно предвидеть и учесть.

В конечном итоге главное — это правильный подход к оценке. Не важно, насколько красивым может получиться конвейер непрерывного развертывания после внедрения конкретных мер, важно, чтобы эти меры давали прибыль с разумным соотношением к предыдущим затратам. Меры, которые не соответствуют этим условиям, не должны реализовываться.

Именно эта идея лежит в основе понятия капиталовложений в качество. Любые затраты и выгоды должны быть очевидными. Концепция капиталовложений в качество выдвигает отношение между этими затратами и выгодами на передний план.

9.4.4. Запрещайте сохранение изменений в случае ошибки сборки!

Следующее простое правило может облегчить внедрение методологии непрерывного развертывания:

Если в конвейере непрерывного развертывания обнаруживается ошибка, никто не должен сохранять свои изменения в репозиторий.

В основе этого правила лежат прежде всего практические причины. Если в настоящий момент система находится в неработоспособном состоянии, дополнительные изменения будут усложнять поиск ошибок. Кроме того, ошибки в новом коде не будут обнаружены немедленно, потому что конвейер непрерывного развертывания не работает. В результате они могут остаться незамеченными.

Это простое правило требует от команды разработчиков постоянно поддерживать конвейер непрерывного развертывания в «зеленом» состоянии, иначе исчезнет возможность переносить изменения в рабочее окружение. Правило будет вынуждать членов команды стремиться оптимизировать его работу, чтобы не останавливать разработку из-за обнаруживаемых ошибок. Конкретная реализация мер по оптимизации конвейера зависит от команды — как вариант, можно выделить человека, который будет координировать действия по устранению текущих проблем.

9.4.5. Останови конвейер

Еще один вариант оптимизации конвейера непрерывного развертывания — реализация принципа «Останови конвейер». Эта идея зародилась в промышленном производстве: каждый рабочий на сборочном конвейере может остановить его, обнаружив проблему, которая должна быть решена до повторного пуска конвейера. Этот принцип можно использовать в контексте непрерывного развертывания: когда в конвейере возникает проблема, все члены команды должны собраться вместе, определить важность проблемы и решить, как ее исправить. Это способствует скорейшему устранению проблемы. Главная выгода от внедрения принципа «Останови конвейер» — немедленное устранение проблем. Ему придается большая важность и уделяется максимальное внимание. Это позволяет гарантировать качественную работу конвейера. Однако найденное решение может устранять конкретную проблему, но ее причины могут оставаться невыявленными.

9.4.6. Пять почему

Для исправления этого недостатка можно воспользоваться идеей «пяти почему» [2]. Согласно ей, причина проблемы выясняется пятью вопросами «Почему?», задаваемыми подряд.

Например:

- Почему сборка потерпела неудачу?
- Потому что тест завершился ошибкой.
- Почему тест завершился ошибкой?
- Потому что для тестирования использовались некорректные данные.
- Почему использовались некорректные данные?
- Потому что возникла ошибка в момент импортирования тестовых данных.
- Почему возникла эта ошибка?
- Потому что тестовые данные создавались вручную.
- Почему тестовые данные создавались вручную?
- Потому что эта операция не была автоматизирована.

В данном примере мерой по устранению проблемы является автоматизация создания тестовых данных. После ее реализации в каждом прогоне конвейера будут участвовать одни и те же тестовые данные и будет устранена причина возникшей проблемы.

В контексте следования принципу «Останови конвейер» в центре внимания находится другая идея: здесь главное — быстро ликвидировать проблему. Чтобы реализовать автоматическое создание тестовых данных, требуется некоторое время. Поэтому в контексте принципа «Останови конвейер» решением будет исправление тестовых данных, а не устранение глубинной проблемы. Да, работа конвейера будет быстро восстановлена. Но проблема может проявиться снова, в последующих прогонах конвейера.

9.4.7. DevOps

Методология непрерывного развертывания требует тесного сотрудничества оперативного персонала и разработчиков. Каждая из этих групп «владеет» своей частью конвейера непрерывного развертывания. Специалисты, занимающиеся эксплуатацией, в полной мере владеют такими аспектами, как мониторинг, безопасность и сетевые инфраструктуры, необходимыми

для установки и работы приложения. Разработчики, в свою очередь, хорошо знают код, инфраструктуру разработки и промежуточное программное обеспечение, такое как сервер приложений. Работая в тесном сотрудничестве, эти два коллектива с легкостью смогут построить конвейер непрерывного развертывания. Поэтому, приступая к внедрению методологии непрерывного развертывания, обязательно следует позаботиться об организации такого сотрудничества (см. главу 10, «Непрерывное развертывание и DevOps»).

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ Набросайте текущий процесс выпуска новых версий в известном вам проекте, используя прием систематизации потока ценностей (см. рис. 9.1 и 9.2).
- ▲ Как долго изменения находятся на каждом этапе?
- ▲ Сколько изменений оказывается в очереди ожидания на каждом этапе потока ценностей?
- ▲ Если эта информация не известна, как ее можно получить?
- ▲ Исходя из полученной информации, предложите меры по оптимизации. Для улучшений всегда найдется место...
- ▲ Предложите пару-тройку мер по оптимизации конвейера непрерывного развертывания, включая анализ затрат/выгод в соответствии с идеей капиталовложений в качество. Какие меры вы реализовали бы в первую очередь?
- ▲ Определите группу, способную выполнить эти оптимизации.
- ▲ Какие навыки необходимы?
- ▲ Смогут ли разработчики или специалисты по эксплуатации самостоятельно реализовать предложенные меры или требуется их сотрудничество?
- ▲ Проведите исследования, руководствуясь принципом «пять почему». Попробуйте применить его к последней проблеме, с которой вам пришлось столкнуться.

9.5. В заключение

Методология непрерывного развертывания может и по возможности должна использоваться с самого начала проекта. Это позволит строить конвейер постепенно, шаг за шагом. В проектах, развивающихся какое-то время, уже имеется конвейер переноса изменений в рабочее окружение. Этот конвейер «достаточно лишь» оптимизировать для соответствия идее непрерывного развертывания. Для этого можно использовать следующие подходы.

- Систематизация потока ценностей помогает проанализировать конвейер и оценить временные задержки на каждом его этапе. Эту информацию можно использовать при принятии решений по оптимизации.
- Метод капиталовложений в качество предполагает оценку затрат на оптимизацию.
- «Останови конвейер» служит защитной мерой, помогающей максимально быстро устранять проблемы, возникающие во время прогона конвейера.
- Чтобы обезопасить себя от проблем в будущем, можно использовать метод «пять почему».

Для строительства и оптимизации конвейера непрерывного развертывания на практике чаще применяются комбинации этих методов.

Ссылки

1. <https://www.infoq.com/articles/no-more-technical-debt>
2. https://ru.wikipedia.org/wiki/Пять_почему

10

Непрерывное развертывание и DevOps

10.1. Введение

Непрерывное развертывание — в первую очередь технологическая методика. В этой главе обсуждаются последствия внедрения методики непрерывного развертывания в организации. В разделе 10.2 описывается интеграция разработки и эксплуатации (DevOps): это форма организации, которая оптимально подходит для реализации непрерывного развертывания. Главной целью DevOps является налаживание сотрудничества службы эксплуатации (Ops) и команды разработчиков (Dev). Раздел 10.3 демонстрирует конкретные примеры взаимодействий в рамках интеграции разработки и эксплуатации для целей непрерывного развертывания. В разделе 10.4 задается вопрос о возможности внедрения непрерывного развертывания без интеграции разработки и эксплуатации. В заключение в разделе 10.5 освещаются дополнительные организационные подходы, расширяющие интеграцию.

10.2. Что такое DevOps?

Классически в организации разработкой и эксплуатацией занимаются разные подразделения, которые пересекаются только на самом высоком уровне иерархии управления ИТ. Они также имеют разные цели: подразделение, занимающееся эксплуатацией, должно работать максимально эффективно, а эффективность оценивается затраченными средствами. Подразделение, занимающееся разработкой, реализует новые возможности, и его работа оценивается по тому, как быстро и эффективно оно вводит эти возможности. Причиной столь резкого разграничения является идея разделения труда и связанная с ней высокая эффективность, обычно достигаемая за счет специализации, стандартизации и индустриализации. Когда подраз-

деление эксплуатации, к примеру, поддерживает только платформу одного типа, оно может специализироваться на этой платформе и осуществлять поддержку особенно эффективно с применением средств автоматизации.

10.2.1. Проблемы

Однако такой подход к организации может также порождать проблемы. Подразделение эксплуатации стремится достигнуть максимальной стабильности программного обеспечения, и каждое изменение, каждая новая версия могут рассматриваться ими как угроза стабильности. Подразделение разработки, напротив, стремится развивать и изменять программное обеспечение. Это легко может привести к конфронтации между подразделениями, в ходе которой может «затеряться» общая цель обоих подразделений — оптимальное обслуживание клиентов, где бы они ни находились: внутри организации или на внешнем рынке.

Также существенны различия в повседневной работе: подразделения разработки и эксплуатации рассматривают приложение с совершенно разных точек зрения. С точки зрения эксплуатации приложение — это процесс в операционной системе, который можно контролировать с помощью соответствующих инструментов мониторинга. Помимо использования процессорного времени и объема ввода/вывода можно также оценить нагрузку на ядро операционной системы. Применяя такие инструменты, служба эксплуатации анализирует поведение приложений и обнаруживает возникающие ошибки. При этом приложение часто остается черным ящиком, внутреннее устройство которого и особенности работы неизвестны. Разработчики, напротив, хорошо знают прикладную логику и работают, например, с исключениями и файлами журналов. Эксплуатационники также должны приобретать эти знания. Часто они не знают конкретных инфраструктур, таких как виртуальная машина Java, и их особенностей, таких как автоматическая сборка мусора, хотя нередко эти знания очень важны для анализа проблем. Разработчики, в свою очередь, часто не знакомы с типичными инструментами и приемами, используемыми в ходе эксплуатации, поскольку они обычно администрируют только свои компьютеры или тестовые серверы, на которых эти инструменты отсутствуют или не могут использоваться во всей их полноте. Кроме того, эти системы едва ли могут сравниться с промышленными системами.

Соответственно, поскольку подразделения эксплуатации и разработки обладают только частью необходимых знаний и инструментов, полно-

ценная эксплуатация приложений возможна только в тесном сотрудничестве. Это особенно актуально для разработки, потому что должен существовать путь для передачи обратной связи от эксплуатации в разработку — именно эту область охватывает методология непрерывного развертывания.

10.2.2. Точка зрения клиента

Такое резкое разграничение на эксплуатацию и разработку также оказывается вредным для клиентов: когда возникает ошибка или обнаруживается проблема, в зависимости от причины она может быть решена службой эксплуатации или разработчиками. Однако клиенту часто трудно определить, какое подразделение может ему помочь в решении проблемы. В худшем случае служба эксплуатации сошлется на подразделение разработки как единственное, которое может устранить проблему, а разработчики могут заявить, что устранение данной конкретной проблемы находится в компетенции службы эксплуатации. Клиенту сложно решить, кто, в конце концов, должен нести ответственность.

10.2.3. Первопроходцы: Amazon

Разделение на эксплуатацию и разработку в основном характерно для крупных ИТ-организаций. Однако в 2006 году стало известно, что одна из по-настоящему крупных ИТ-компаний — Amazon — реализовала иной подход. Начиная с 2006 года в Amazon появились коллективы, осуществляющие не только разработку, но и эксплуатацию. Каждый коллектив отвечает за определенную службу, реализующую прикладную логику. То есть каждый коллектив может свободно оптимизировать свою службу так, чтобы не усложнить ее эксплуатацию и дальнейшее развитие. Например, не тратя времени на согласования, коллектив может внедрить расширенные средства мониторинга приложения или самостоятельно решить, какой стек технологий использовать, — в конце концов, им внедрять и эксплуатировать эти технологии, а также устранять появляющиеся ошибки и проблемы. При таком подходе устраняются практически все общеорганизационные ограничения, кроме одного — все приложения должны выполняться в инфраструктуре Amazon Cloud. На первый взгляд такой способ организации должен порождать хаос, но в действительности он дает большую свободу и обеспечивает основу для самоорганизации.

Остается лишь узкий круг лиц, занимающихся исключительно эксплуатацией, независимо от других коллективов, и отвечающих за обслуживание и поддержку аппаратного обеспечения и инфраструктуры Amazon Cloud. Остальные коллективы устанавливают операционные системы и другое программное обеспечение на свои виртуальные машины.

10.2.4. DevOps

Термин DevOps был предложен в 2009 году на конференции Devopsdays в Бельгии. Он состоит из двух частей: разработка (Development, Dev) и эксплуатация (Operations, Ops). Этот термин отражает основную идею: разработка и эксплуатация должны осуществляться одним коллективом, делящимся на подразделения по признаку области ответственности. Для клиентов такая организация IT-услуг более понятна: если возникает проблема с конкретной службой, сразу очевидно, к кому можно обратиться для ее исправления. Кроме того, клиенту ясно, кто сможет расширить конкретную службу и добавить в нее дополнительные функции.

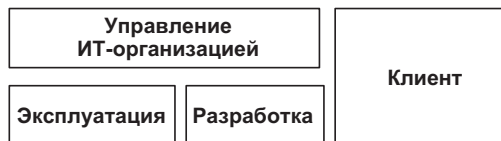


Рис. 10.1. Классическая IT-организация с разделением на разработку и эксплуатацию

То есть целью интеграции разработки и эксплуатации (DevOps) является замена организационной модели (рис. 10.1). Отдельные структурные единицы, такие как службы эксплуатации и разработки, объединяются для внедрения комплексного подхода, в котором каждый коллектив берет на себя ответственность за обе области.

То есть, когда разработка и эксплуатация осуществляются одним коллективом, отпадает необходимость создавать еще один коллектив, который играл бы роль связующего звена между разработкой и эксплуатацией, так как создание дополнительного, третьего подразделения противоречит цели сокращения структурных подразделений.

Гораздо полезнее реализовать и запустить службу, с самого начала опираясь на идею DevOps. Эту службу будет поддерживать единый коллектив,

одновременно занимающийся разработкой и эксплуатацией. Это может стать первым семенем DevOps, из которого вырастет иная организационная структура (рис. 10.2).

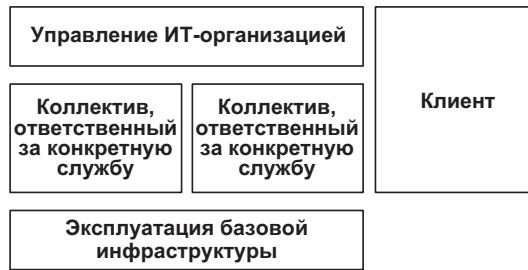


Рис. 10.2. Организация DevOps

На практике зачастую очень сложно изменить всю структуру организации, особенно на крупных предприятиях. Однако существуют иные пути налаживания сотрудничества: так или иначе, в дополнение к формальной организационной структуре имеются и неформальные пути сотрудничества. Эту тенденцию можно поощрять — например, размещая работников, занимающихся эксплуатацией, в одном помещении с разработчиками. Это упростит их общение друг с другом и, как результат, будет способствовать тесному сотрудничеству. Кроме того, сотрудников, занимающихся эксплуатацией, можно направлять на стажировку в коллектив разработчиков и наоборот. Это также поспособствует налаживанию обмена знаниями и сотрудничеству без формального изменения организационной структуры.

В конце концов, DevOps представляет иную идею организации разработки и эксплуатации. Оба подразделения должны тесно взаимодействовать друг с другом, это позволит создать максимальную ценность для клиента. Поэтому изменение организационной структуры, хотя и не является обязательным, будет безусловно полезным.

10.3. Непрерывное развертывание и DevOps

Непрерывное развертывание легче реализуется в контексте DevOps, потому что требует знаний, которыми обладают сотрудники двух подразделений: разработчики знают приложение — как оно устроено, как настра-

ивается и какие метрики представляют особый интерес для мониторинга с точки зрения логики работы приложения. Работники, занимающиеся эксплуатацией, в свою очередь, знают окружающие условия и могут, например, предоставить инструменты для мониторинга или анализа журналов. Благодаря интеграции DevOps объединенная команда способна охватить сразу обе области: разработку и эксплуатацию. Кроме того, обратная связь, полученная из рабочего окружения, может немедленно использоваться для оптимизации приложения, потому что все необходимые специалисты работают в одной команде. Это позволяет достичь главной цели методологии непрерывного развертывания — быстрого получения обратной связи.

10.3.1. DevOps: не только непрерывное развертывание

Многие считают понятие «непрерывное развертывание» синонимом DevOps. Однако это не соответствует действительности. Несмотря на то что реализация непрерывного развертывания существенно упрощается за счет DevOps и представляет собой одну из базовых методик в области DevOps, существует много других сфер кроме непрерывного развертывания, где DevOps оказывается весьма полезным организационным решением.

- Например, разработка и эксплуатация могут совместно работать над реализацией мониторинга. Эксплуатационники осуществляют мониторинг процессов и самого сервера на уровне операционной системы. Разработчики делают выводы по результатам мониторинга, оптимизируют приложение и, соответственно, повышают его надежность. Кроме того, разработчики могут предлагать дополнительные метрики для мониторинга, а эксплуатационники — включать их в мониторинг. Это даст команде возможность получить дополнительные сведения о поведении приложения. В мониторинг могут также включаться прикладные метрики. Если выпуск новой версии повлечет падение продаж, это будет сразу заметно, благодаря чему проблема может быть быстро исправлена.
- Кроме того, эксплуатация и разработка могут вместе работать над устранением неполадок. Несмотря на то что эксплуатационники имеют большой опыт в анализе неисправностей и проблем, они часто ограничиваются системными инструментами, такими как `tcpdump` или `strace`. Они смотрят на приложение со стороны. Разработчики знают внутреннее устройство приложения и, соответственно, обладают более полной

информацией, необходимой для устранения неполадок. То есть они могут расширить приложение так, чтобы оно возвращало дополнительную информацию, или даже встроить в него специализированные инструменты анализа. Например, инструменты профилирования и трассировки разных уровней приложения. Реализация может быть очень простой — например, можно предусмотреть в URL дополнительный параметр, получив который приложение будет отображать на странице дополнительные данные. Кроме того, могут быть предусмотрены дополнительные элементы управления для нужд эксплуатации — например, для отключения разных частей приложения.

- Наконец, разработчики могут дать эксплуатационникам возможность вмешательства в приложение. Обычно, кроме перезапуска приложения, эксплуатационники не имеют возможности как-то иначе вмешаться в его работу. Однако возникает необходимость отключить некоторые функции — например, когда другая система, используемая для работы этих функций, временно недоступна или должна быть остановлена для обслуживания. Это может помочь избежать аварийного завершения всего приложения. Кроме того, для исправления ошибок в наборах данных может пригодиться административный инструмент для их изменения. Конечным результатом может стать инструмент администрирования приложения, позволяющий администратору устранять проблемы, возникающие в приложении.

То есть совершенно очевидно, что DevOps — это намного больше, чем непрерывное развертывание. Это особая точка зрения и форма организации, которая может привести к внедрению многих технических мер, и методология непрерывного развертывания — лишь одна из них.

10.3.2. Индивидуальная ответственность и самоорганизация

По сути, DevOps предполагает индивидуальную ответственность. Команды берут на себя всю ответственность за свои компоненты — за их разработку и эксплуатацию. В результате им приходится тратить намного меньше времени на координирование своих действий с другими командами. Все, что связано с конкретным компонентом, может осуществлять единая команда, включая разработку, эксплуатацию и, конечно же, развертывание новых версий в рабочем окружении. Это позволяет работать быстрее, быстрее разрабатывать программное обеспечение и быстрее внедрять изменения. Например, отпадает необходимость писать пространные руководства по

эксплуатации, потому что эксплуатационники и разработчики компонента работают рука об руку. Обмен письменными документами во многих случаях заменяется прямым общением. Кроме того, благодаря автоматизации уменьшается количество ручных операций, которые должны документироваться в обязательном порядке.

10.3.3. Технологические решения

Более того, такой подход позволяет командам самостоятельно принимать много нужных решений. Они могут выбрать для использования новую технологию без обращения к руководству. Однако это также означает, что команда берет на себя ответственность и за эксплуатацию этой технологии. Если выбранная технология вызовет проблемы в рабочем окружении, команде придется самой искать пути решения. Именно поэтому команда принимает на себя всю ответственность за собственные решения. Как следствие, она должна организовать дежурство по вызову. Поэтому в интересах всех членов команды — разработчиков и эксплуатационников — избегать проблем, которые могут заставить подниматься и бежать на работу среди ночи.

10.3.4. Меньше централизованного управления

Централизованное управление менее важно в такой среде. Конечно, должны иметься и выполняться некоторые общие правила. Но эти общие правила должны касаться базовых аспектов: каждая команда имеет конвейер непрерывного развертывания и, соответственно, использует автоматизированную процедуру развертывания компонентов в разных окружениях, включая автоматическое тестирование. Однако решение о выборе конкретных технологий делегируется отдельным командам. Выбор технологий реализации, таких как фреймворки, языки программирования или серверы приложений, полностью перекладывается на команды, потому что именно они несут ответственность за разработку и эксплуатацию. Если команда выбирает технологию, лучше соответствующую ее требованиям, нет причин отвергать этот выбор. В конце концов, именно эта команда, а не отдельное подразделение эксплуатации будет решать проблемы, если выбор окажется неудачным. Аналогично, команда должна нести ответственность, если выбранная технология повлечет большие затраты на реализацию.

10.3.5. Плюрализм технологий

Это позволяет использовать много разных технологий в пределах одной организации. Классическая организация стремится ограничить количество технологий и контролировать их использование с целью минимизации риска и реализации синергетического потенциала. Если все коллективы используют одни и те же языки программирования и инфраструктуру, каждый разработчик будет владеть необходимыми технологиями и сможет поддержать любой проект, до определенной степени, разумеется. Эксплуатационники получают возможность сосредоточиться на овладении узким кругом технологий и ознакомиться с проблемами, характерными для них.

Такой синергетический эффект не является целью DevOps. Главная цель — свобода принятия решений командами. То есть каждая команда может выбрать свой стек технологий, лучше соответствующих конкретным требованиям. Это позволяет каждой команде достичь оптимальной производительности труда.

10.3.6. Обмен специалистами между командами

Обмен специалистами между командами способствует обмену опытом и самостоятельно разработанными инструментами или фреймворками. Для этого можно даже утвердить стандартный стек технологий. Но принуждение к использованию этого стека окупается преимуществами предлагаемых технологий: другие команды, уже имеющие опыт использования этих технологий, смогут в этом случае оказать поддержку. Кроме того, подобные команды могут иметь самостоятельно разработанные инструменты, что сделает технологию, используемую одной группой, более привлекательной для других. В конечном итоге, несмотря на свободу выбора, высока вероятность того, что другие команды также примут предлагаемые технологии на вооружение.

Также можно несколько ограничить полную свободу выбора: если определенной технологией владеет только один член команды, могут возникнуть проблемы, если этот человек уйдет в отпуск или уволится из компании. Чтобы предотвратить такие ситуации, можно установить правила, допускающие использование технологии, если только в команде имеется несколько человек, владеющих ею. Конечно, эта проблема существует также на уровне команды: когда команда использует технологии, неизвестные ни-

кому другому в организации, необходимо принять все меры, чтобы обеспечить долгосрочное существование такой команды. Наконец, порой сложно перевести человека из одной команды в другую, чтобы обеспечить дополнительную поддержку на время, потому что члены одной команды могут не владеть стеком технологий, используемых другой командой.

10.3.7. Архитектура

С позиции строительства архитектуры высокая степень независимости команд представляет определенную проблему: требование центральной инстанции развивать архитектуру в определенном направлении, противоречит независимости команд. Его необходимо заменить процессом, который позволил бы командам продолжать развивать систему скоординированным образом. Этот процесс может возникнуть спонтанно, но, как учит опыт, желательно, чтобы он был управляемым — необходимо проводить совещания, на которых команды могли бы координировать свои действия. Также может потребоваться, чтобы главный архитектор задавал основные направления развития архитектуры — например, опираясь на планы команд по дальнейшему развитию — и осуществлял процесс координации между командами. При этом важно, чтобы решения по основным направлениям принимались командами, а центральная инстанция лишь управляла этим процессом. В конечном итоге именно командам придется воплощать в жизнь архитектурные решения, и они должны иметь возможность разрабатывать программное обеспечение максимально эффективно.

То есть интеграция разработки и эксплуатации (DevOps) влияет на процессы, команды и подходы не только в пределах непрерывного развертывания. В то же время DevOps дает множество дополнительных преимуществ. Конечно, непрерывное развертывание можно реализовать без организационных изменений, связанных с интеграцией DevOps, но они совершенно необходимы для получения всех выгод, предлагаемых конвейером непрерывного развертывания. То есть такой конвейер нельзя реализовать в полном объеме без определенного уровня кооперации и координации.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

Если вы являетесь разработчиком в классической организации, пообщайтесь с коллегами из службы эксплуатации, которые обслуживают ваше приложение, и выясните:

- ▲ Как осуществляется мониторинг?
- ▲ Какие манипуляции они обычно выполняют с приложением?
- ▲ Какие инструменты используют для этого?
- ▲ Какие возможности могли бы помочь им в устранении неполадок?
- ▲ Какие инциденты возникают и как часто им приходится приходить на работу по ночам? Каковы причины этих инцидентов?
- ▲ Как выглядит первая десятка самых частых ошибок, встречающихся в файлах журналов?
- ▲ Какие наибольшие сложности возникают при установке и эксплуатации приложения?

Если вы работаете эксплуатационником в классической организации, узнайте, кто разрабатывал приложение, и обсудите с ним вопросы, перечисленные выше.

- ▲ Выберите приложение как пример для мониторинга.
- ▲ Какие параметры можно извлечь из приложения в данный момент?
- ▲ К какой категории они относятся — например, технические (база данных, виртуальная машина), прикладные (объем продаж, количество регистраций новых клиентов)?
- ▲ Для каких параметров определены сигналы тревоги?

10.4. Непрерывное развертывание без DevOps?

Как отмечалось в предыдущих разделах, внедрение непрерывного развертывания дает наибольшие выгоды в сочетании с организацией DevOps, потому что для реализации разных этапов конвейера непрерывного развертывания необходимы навыки разработки (Dev) и эксплуатации (Ops). С другой стороны, внедрение интеграции DevOps в большинстве организаций невозможно без существенных структурных изменений. Особенно это относится к крупным предприятиям, где области разработки и эксплуатации часто разделены вплоть до уровня ИТ-директора. Для внедрения интеграции DevOps организации пришлось бы коренным образом изменить свою структуру, чтобы обеспечить создание единых команд из разработчиков и эксплуатационников. Однако такие коренные измене-

ния требуют детальной проработки и часто порождают активное сопротивление.

Поэтому возникает вопрос: можно ли внедрить методологию непрерывного развертывания без DevOps. Для реализации непрерывного развертывания не требуется наличие смешанных команд, но необходима совместная работа при создании конвейера. Выражаясь конкретнее, эксплуатация должна автоматизировать развертывание, а разработка — сосредоточиться на реализации разных видов тестирования. Взаимная поддержка сверх этого не только допустима, но и приветствуется. Такая совместная работа над созданием конвейера не требует реструктуризации организации.

Однако организационные границы часто находят отражение в конвейере: например, разработчики могут настроить тестовые системы и автоматизировать их. Однако, если эксплуатационники пожелают не рисковать и предпочтут реализовать собственную автоматизацию, конвейер распадется на две части. Системы и автоматизация в подразделении разработки будут структурированы иначе, чем в подразделении эксплуатации. Это затруднит воспроизведение в окружении для разработки тех проблем, которые возникают в рабочем окружении, а также удвоит затраты, потому что изменения придется отслеживать в обеих системах. Это верно, даже если разработка и эксплуатация используют одни и те же инструменты и эксплуатация предпочитает переносить изменения из разработки вручную. Оба подхода к автоматизации будут развиваться отдельно друг от друга, и последующая унификация потребует больше усилий. Эта техническая проблема обусловлена организационной проблемой: эксплуатация не доверяет результатам тестирования, проведенного на стороне разработки, и поэтому не торопится принимать изменения. Однако данная проблема также решается без реструктуризации организации, простым сотрудничеством подразделений. После преодоления первоначального недоверия легко находится техническое решение. С другой стороны, никакое техническое решение не устранит психологическую проблему.

10.4.1. Завершение конвейера непрерывного развертывания

В отсутствие поддержки со стороны эксплуатации возможность организации непрерывного развертывания вообще становится маловероятной. Конечная цель непрерывного развертывания — скорейшее внедрение программного обеспечения в эксплуатацию. Это практически невозможно,

если эксплуатация не предпринимает усилий по его реализации и поэтому конвейер не может проникать в рабочее окружение. Тем не менее все еще существует возможность реализовать конвейер с разными этапами тестирования и закончить его на границе с рабочим окружением. Однако в этом случае отсутствует возможность достижения главной цели — быстрой передачи изменений в эксплуатацию.

Однако скорость вывода на рынок — не единственная цель методологии непрерывного развертывания. Другими ее целями являются воспроизводимость и повторяемость. Обе они вполне могут быть достигнуты с укороченным конвейером непрерывного развертывания. Все тесты останутся воспроизводимыми. Кроме того, тестирование повторяется с каждым изменением, поэтому они могут выполняться намного чаще. Это способствует увеличению качества программного обеспечения и ускорению обратной связи — важнейшей цели непрерывного развертывания. Помимо этого, программное обеспечение как минимум потребуется устанавливать в тестовых системах, а значит, и процесс установки также будет в принципе воспроизводимым.

Повышение качества программного обеспечения и автоматизация, а значит, и воспроизводимость его развертывания, по крайней мере в тестовых системах, уже достаточно веская причина для внедрения методологии непрерывного развертывания. То есть можно не только внедрить непрерывное развертывание без DevOps, но и реализовать укороченный конвейер без поддержки со стороны эксплуатации.

К сожалению, в этом случае не удастся получить все преимущества непрерывного развертывания, но это по-прежнему весьма полезный шаг. Кроме того, эксплуатация, со своей стороны, может продолжить конвейер непрерывного развертывания позднее.

Конвейер непрерывного развертывания, который реализуется только подразделением эксплуатации, будет включать лишь автоматическое развертывание в рабочее окружение. Эксплуатации в любом случае придется реализовать автоматизацию, чтобы ограничить расходы. Однако быстрое развертывание в рабочем окружении возможно только при наличии этапов тестирования, поскольку только в этом случае можно убедиться, что программное обеспечение достаточно качественное для развертывания. Даже если процесс развертывания полностью автоматизирован и надежен, никто не будет внедрять новое программное обеспечение без предварительного тестирования, потому что только тестирование гарантирует необходимое качество.

В целом можно реализовать непрерывное развертывание без интеграции DevOps, но конечный эффект будет ниже, чем при внедрении в сочетании с DevOps.

10.5. В заключение

Интеграция DevOps распространяет непрерывное развертывание на организационный уровень, обеспечивая более тесное сотрудничество разработки и эксплуатации. Главной целью непрерывного развертывания является внедрение программного обеспечения. Для этого сотрудники двух подразделений должны тесно сотрудничать, поэтому интеграция DevOps особенно положительно сказывается на реализации непрерывного развертывания. Однако сотрудничество необходимо не только для этого, но также для организации мониторинга и устранения неполадок. Следующим шагом могло бы стать создание команд с еще более широкими полномочиями, как предлагает методика Design Thinking (Дизайн-мышление) [1]. Метод Lean Startup [2] предлагает оценивать особенности для поэтапного улучшения продукта. Так как непрерывное развертывание позволяет развертывать и оценивать отдельные функции, подход Lean Startup может быть воплощен вместе с непрерывным развертыванием.

Ссылки

1. https://en.wikipedia.org/wiki/Design_thinking
2. Eric Ries: *Kanban: The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Business, 2010, ISBN 978-0-67092-160-7 (Эрик Рис. Бизнес с нуля. Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели. М.: Альпина Паблишер, 2016. ISBN: 978-5-9614-6028-5. — Примеч. пер.).

11

Непрерывное разворачивание, DevOps и архитектура ПО

11.1. Введение

Непрерывное разворачивание и интеграция DevOps — это подходы, позволяющие эффективнее разрабатывать программное обеспечение и внедрять его в эксплуатацию. Они в основном влияют на процессы разворачивания и эксплуатации. Однако в этой главе основное внимание уделено совершенно иному аспекту: влиянию непрерывного разворачивания на архитектуру приложений. На первый взгляд эта связь неочевидна — почему приемы разворачивания и эксплуатации программного обеспечения должны влиять на его архитектуру?

В первую очередь в разделе 11.2 определяется суть термина «архитектура программного обеспечения». В разделе 11.3 обсуждается, как деление программного обеспечения на компоненты можно оптимизировать для реализации непрерывного разворачивания. В разделе 11.4 основное внимание уделяется интерфейсам между компонентами, а в разделе 11.5 демонстрируется, как правильно организовать поддержку баз данных, что является довольно сложной задачей в контексте непрерывного разворачивания. Главной целью методологии непрерывного разворачивания является внедрение новых особенностей; именно этой цели посвящен раздел 11.6. В разделе 11.7 рассказывается, как реализовать новые особенности в архитектуре программного обеспечения. Наконец, в разделе 11.8 дается заключительный вывод.

11.2. Архитектура программного обеспечения

Архитектуру программного обеспечения [1] можно определить как совокупность важнейших решений, касающихся организации программной системы. К их числу относятся также решения о делении системы на от-

дельные компоненты, установлении связей между ними и определении особенностей компонентов и связей. Это очень общее определение. В зависимости от уровня, на котором рассматривается архитектура программного обеспечения, можно привести разные примеры компонентов и связей.

- В объектно-ориентированной системе роль компонентов могут играть классы. В этом случае под связями между компонентами подразумеваются способы использования одних классов другими или отношение наследования.
- Языки программирования, такие как Java, C# или C++, предлагают возможность организации связанных классов в пакеты или пространства имен. Эти структурные единицы также используются для реализации компонентов. Связями в этом случае являются операции использования классов из других пакетов или пространств имен.
- На более высоком уровне компонентами могут быть единицы развертывания, роль связей между ними играют зависимости между отдельными единицами. Примерами таких единиц развертывания являются файлы WAR или EAR для Java, динамические библиотеки DLL для .NET и библиотеки для других систем.
- На уровне архитектуры корпоративного программного обеспечения отдельными компонентами могут быть программные системы. В этом случае роль связей играют вызовы одной системы другой.

Основополагающим решением при организации архитектуры программного обеспечения является техническая реализация компонентов — то есть как архитектурные решения проявляют себя в системе.

11.2.1. Зачем нужна архитектура программного обеспечения?

Архитектура программного обеспечения играет важную роль, поскольку охватить программную систему целиком порой очень непросто. Поэтому систему делят на компоненты. Разработчикам проще разбираться в таких компонентах по отдельности и изменять их. Связи между компонентами также относительно просты для понимания. Только если разработчик понимает эти аспекты, он может изменять компоненты или связи между ними и таким способом менять систему.

Кроме того, архитектура определяет технические основы системы. В конце концов, должны быть реализованы все ее отдельные архитектурные компо-

ненты, а для этого требуется принимать определенные технические решения. Например, объектно-ориентированные системы могут быть реализованы на разных языках, и для решения типовых задач, таких как хранение данных, могут использоваться разные библиотеки. Эти решения формируют стек технологий и влияют на нефункциональные требования, такие как производительность, защищенность и масштабируемость.

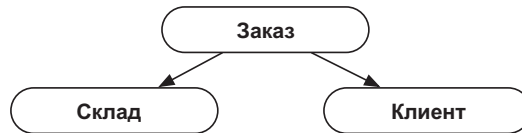


Рис. 11.1. Деление системы на компоненты

В качестве примера рассмотрим очень простую систему, которая передает заказы предприятию (рис. 11.1). Для этого требуется три компонента.

- Компонент, представляющий заказ, отвечающий за реализацию процесса оформления заказа и дополнительно предоставляющий доступ к заказам, обработанным к настоящему моменту.
- Компонент, представляющий склад, хранит список имеющихся товаров и их количество. Может также запускать процедуру доставки товара клиенту.
- Компонент, представляющий клиента, хранит данные о клиенте и передает их для обработки заказа.

Деление на компоненты выполнено исключительно по границам ответственности и никак не зависит от выбора технологий. Этот подход имеет большое значение, потому что каждая программная система должна реализовать некоторые функции. Организация архитектуры в соответствии с границами функций обеспечивает поддержку фактической цели: разработка программной системы для реализации определенных возможностей. Иногда эта цель теряется в процессе определения архитектуры, и на передний план выводятся используемые технологии. Изначальная независимость архитектуры от технологий помогает избежать этой проблемы.

Конечно, технический аспект реализации компонентов никуда не исчезает: компоненты могут быть классами, пакетами или единицами развертывания. Однако это решение не зависит от решения о разделении функций между ними. То есть всю систему можно организовать в виде единствен-

ной единицы развертывания и реализовать отдельные компоненты как классы или пакеты. Этот подход имеет много преимуществ: он технически прост и гарантирует высокую производительность, поскольку компоненты взаимодействуют друг с другом непосредственно, через вызовы методов друг друга; соответственно, отсутствуют накладные расходы на распределенные взаимодействия. Если бы каждый компонент был отдельной единицей развертывания, проект получился бы намного сложнее, особенно в отношении процесса сборки, поскольку каждая единица развертывания обычно оформляется в виде отдельного проекта, который должен отдельно компилироваться и так же отдельно устанавливаться в рабочем окружении. Кроме того, компонентам в этом случае часто приходится использовать сложные механизмы взаимодействий — например, SOAP или REST. Это не только усложняет реализацию, но также ухудшает производительность.

На первом этапе все компоненты можно включить в одну, общую единицу развертывания ради уменьшения сложности и достижения высокой производительности. Взаимодействия между компонентами реализуются в этом случае как вызовы методов (рис. 11.2).

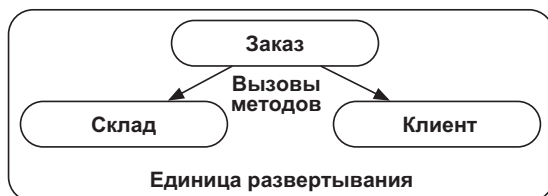


Рис. 11.2. Реализация компонентов

11.3. Оптимизация архитектуры для непрерывного развертывания

Однако если изменяется только один компонент — например, реализующий обработку заказов, — должен выполняться весь конвейер непрерывного развертывания. В процессе выполнения конвейер компилирует, собирает и тестирует не только изменившийся компонент, но и все остальные. Это приводит к выполнению большого объема ненужной работы и, что особенно важно, возникновению задержки в получении обратной связи с результатами тестирования. Перед приемочными испытаниями компонен-

та, осуществляющего обработку заказов, выполняется этап выпуска всех компонентов; то есть выполняются все модульные тесты всех компонентов, хотя в действительности изменился только один из них. Как результат ошибка на этапе приемочных испытаний будет выявлена только после завершения всех модульных тестов.

Тем не менее такой порядок вещей имеет право на существование: в процессе работы конвейера должны быть протестированы все компоненты. В конце концов, существует вероятность, что ошибка в одном компоненте проявится только в виде ошибочного поведения другого компонента. Однако в случае с компонентом, обрабатывающим заказы, эту вероятность можно исключить, потому что никакие другие компоненты не используют их, как демонстрирует диаграмма на рис. 11.3.

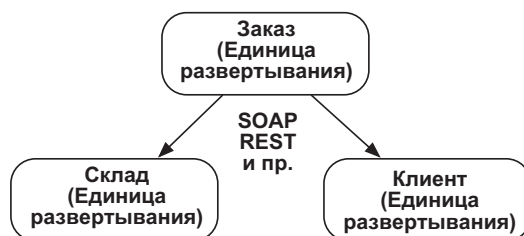


Рис. 11.3. Деление системы на несколько единиц развертывания

11.3.1. Деление на мелкие единицы развертывания

Если требуется учесть этот фактор в архитектуре и оптимизировать ее для непрерывного развертывания и достижения быстрой обратной связи, разумнее использовать иную техническую реализацию компонентов. Каждый компонент может быть реализован как отдельная единица развертывания, например в виде библиотеки, файла JAR, файла WAR или выполняемого приложения на произвольном языке программирования.

В этом случае для каждого компонента — заказы, клиенты и склад — создается отдельный конвейер непрерывного развертывания. Соответственно, при изменении компонента, осуществляющего обработку заказов, запустится только один конвейер, и обратная связь будет получена намного быстрее.

Кроме всего прочего, такой подход ведет к уменьшению рисков: вместо всей системы развертывается только ее часть. То есть объем изменений

уменьшается, что уменьшает риск возникновения ошибки. Помимо этого, развертывание становится проще и выполняется быстрее, потому что размер единиц развертывания намного меньше. По той же причине изменения, произведенные единицей развертывания, откатываются намного быстрее, если вдруг будет обнаружена ошибка.

Этот подход может также влиять на способ взаимодействий компонентов друг с другом: если единицы развертывания являются библиотеками, взаимодействия все еще могут осуществляться посредством вызова методов. Однако в этом случае после развертывания обновленного компонента обычно требуется перезапустить другие компоненты, чтобы загрузить новую версию библиотеки. Но если единицы развертывания превратить в отдельные службы, выполняющиеся в отдельных процессах на серверах, придется также изменить порядок взаимодействий между компонентами. Эта задача имеет несколько решений.

- На основе протокола REST [2]. Он использует протокол HTTP не только для чтения данных, но также для создания новых и изменения существующих записей. В качестве формата данных можно использовать JSON [3]. Но формат ProtoBuf [4] намного эффективнее с точки зрения производительности и размеров передаваемых сообщений. Еще одна альтернатива — XML [5]. Даже при том что этот формат менее эффективен, он поддерживает богатую систему типов и преимущественно используется в контексте предприятия.
- Альтернативой протоколу REST может служить прием асинхронных взаимодействий посредством промежуточного программного обеспечения, ориентированного на обработку сообщений (Message Oriented Middleware, MOM), используемый во многих решениях интеграции, таких как сервисные шины предприятия (Enterprise Service Buses). Это еще больше ослабляет зависимость компонентов друг от друга.

В конце концов, для непрерывного развертывания очень выгодно, если создаются единицы развертывания минимального размера.

11.4. Интерфейсы

Если компонент реализует интерфейсы — например, для связи компонента склада или компонента клиента с компонентом заказа, — в процессе развертывания возникает дополнительная проблема: при изменении интерфей-

са необходимо заново развернуть все зависимые компоненты. Зависимые компоненты, использующие интерфейс, должны следить за изменениями, чтобы надлежащим образом использовать новую версию интерфейса. В результате должен развертываться не один, а сразу несколько компонентов. То есть мы вновь вернулись к проблеме, которую решало деление системы на мелкие единицы развертывания: развертывание становится сложным и рискованным.

Для решения этой проблемы требуется организовать поддержку старой версии интерфейса после его изменения. В этом случае зависимый компонент сможет по-прежнему использовать старый интерфейс, благодаря чему отпадает необходимость развертывать все компоненты сразу; их можно развертывать постепенно. Сначала развертывается компонент с изменившимся интерфейсом. Затем все остальные компоненты, использующие интерфейс, постепенно переводятся на новую версию и развертываются. По окончании процесса перехода на новый интерфейс поддержка старого интерфейса прекращается. Разумеется, поддержка дополнительной версии интерфейса требует больше сил. Однако, учитывая сложность процедуры развертывания и количество компонентов, участвующих в ней в противном случае, данная альтернатива выглядит привлекательнее.

В распределенных системах управление версиями интерфейсов не является чем-то необычным. Для этого, например, номер версии можно хранить как часть URL ресурса REST. Конечно, поддержка старой версии интерфейса прекращается только в том случае, когда не осталось компонентов, использующих ее. Нередко это превращается в большую проблему: иногда неизвестно, какие компоненты используют ту или иную версию интерфейса. Однако в сценарии непрерывного развертывания эта проблема имеет решение: зачастую все компоненты принадлежат одному приложению, за которое целиком отвечает один коллектив. Соответственно, коллектив контролирует все компоненты и может ограничить поддержку старых интерфейсов на разумном уровне — например, поддерживать только текущую и предыдущую версии. Поскольку конвейер непрерывного развертывания помогает уменьшить усилия, необходимые для внедрения компонентов, и сопутствующие риски, легко можно развернуть компоненты заново после перевода их на использование новой версии интерфейса другого компонента. Обычно такие изменения оказываются небольшими, что еще больше уменьшает риски. Фактически в распределенных системах старые интерфейсы продолжают поддерживаться только потому, что никто не хочет рисковать и заново развертывать компоненты исключительно ради перехода на использование нового интерфейса.

Разумеется, это также означает, что изменения в интерфейсах для внешних компонентов, разрабатываемых другими коллективами, или в публичных интерфейсах, доступных из Интернета, должны осуществляться с большой осторожностью: пользователи этих интерфейсов не зависят от коллектива, и поэтому нельзя просто так прекратить поддержку какой-то старой версии интерфейса.

11.4.1. Закон Постела, или принцип надежности

Представленная идея управления версиями интерфейсов в действительности является частным случаем закона Постела, который также известен как принцип надежности [6]. В нем говорится, что компоненты должны быть максимально строгими к своим действиям и снисходительными к действиям других. Иными словами, каждый компонент должен строго следовать установленным правилам при использовании других компонентов, но прощать ошибки при использовании его другими компонентами. Следование принципу надежности улучшает совместимость компонентов: точное соответствие каждого компонента требованиям других компонентов само по себе гарантирует его совместимость, а при наличии каких-то отклонений используемый компонент попытается приспособиться к ним.

Применительно к управлению версиями интерфейсов этот принцип означает, что каждый компонент должен принимать запросы к старой версии интерфейса, но использовать только текущие версии интерфейсов других компонентов.

11.4.2. Страховка от сбоев

Даже при наличии поддержки нескольких версий интерфейсов остается еще несколько проблем, требующих решения. На время обновления компонент может быть отключен. Другие компоненты, вызывающие его, должны обрабатывать подобную ситуацию.

Существуют различные способы выхода из ситуации, когда какой-то компонент оказывается недоступным.

- Использовать значения по умолчанию. Если, например, оказывается недоступным компонент, возвращающий рекомендации для клиента, выводятся общие рекомендации.

- Вместо оригинального алгоритма используется упрощенная версия. Например, если нельзя определить кредитный рейтинг клиента, можно использовать упрощенный алгоритм, допускающий выдачу кредита до определенной фиксированной суммы. Фактически это является бизнес-решением: отказаться от продаж в случае простоя каких-то компонентов или допустить определенную степень риска?
- Еще одно решение — использовать Circuit Breaker [7]. Когда обращение к другому компоненту завершается неудачей, последующие вызовы больше не пересылаются этому компоненту, то есть Circuit Breaker не передает вызовы компоненту, а сразу отвечает, возвращая соответствующее сообщение об ошибке. Таким способом исключаются периоды ожидания ответа от недоступного компонента. Кроме того, такой подход предотвращает накопление чрезмерно большого количества обращений к компоненту. Если такое накопление не предотвратить, компонент столкнется с повышенной нагрузкой, когда он вновь включится в работу, что может вызвать его сбой. Circuit Breaker можно использовать в комплексе с другими мерами, чтобы он не только посылал сообщения об ошибках, но максимально компенсировал недоступность компонента.

То есть идея состоит в том, чтобы выбрать простейшую альтернативу обработки ситуации недоступности компонента. Такой подход поможет изолировать отключенный компонент, например, на время его развертывания.

Это сделает систему более устойчивой. Всегда следует учитывать, что компоненты не всегда доступны и иногда они могут просто не отвечать на запросы. Это особенно важно для распределенных систем, поскольку может выйти из строя сеть или возникнуть другие проблемы с оборудованием, влекущие отключение компонентов.

11.4.3. Состояние

Другая проблема, которая возникает при развертывании новой версии компонента, — его состояние, хранящееся в памяти. Обычно во время обновления состояние компонента теряется. Поэтому для поддержки непрерывного развертывания желательно, чтобы компоненты не имели никакого состояния. Точнее говоря, они не должны иметь состояния, хранящегося в памяти. В конце концов, обслуживание состояния является центральной частью большинства систем, и система вообще без состоя-

ния просто немыслима. Однако это состояние должно храниться во внешней системе, например в базе данных или в кэше. Конечно, кэш постоянно меняется, но это всего лишь один случаев хранения состояния компонента в памяти.

Кстати, эту проблему можно решить посредством инфраструктуры. Например, некоторые веб-серверы способны сохранять состояние. В Java используется термин HTTP-сеанс. Однако информацию из HTTP-сеанса можно сохранять в базах данных или в кэшах, например в memcached, без изменения программного кода. Конечно, новая версия программного обеспечения должна поддерживать извлечение информации о состоянии, сохраненной прежней версией.

11.5. Базы данных

Предположим, что состояние системы сохраняется в базе данных. Поддержка баз данных представляет определенную проблему для сценария непрерывного развертывания. Если структуры данных в программном обеспечении претерпели изменения, необходимо также изменить схему базы данных — например, добавить или удалить столбцы. Для этого необходимо внести соответствующие изменения в данные, уже хранящиеся в базе. Такие изменения потенциально могут затрагивать большие объемы данных. Это усложняет реализацию изменений. Кроме того, подобные изменения очень трудно откатить. Это связано с тем, что изменения в базах данных обычно затрагивают очень большие объемы данных, и подобные изменения требуют значительного времени или могут оказаться вообще невозможными при некоторых обстоятельствах. Большая продолжительность внесения изменений обусловлена самой природой вещей. Помимо этого, такие изменения с трудом поддаются тестированию, потому что для этого требуется база данных, сопоставимая с рабочей базой данных. А создание такой базы данных для тестирования требует немалых затрат, при том что необходимые программное и аппаратное обеспечение дороги.

Программное обеспечение легко изменить и развернуть, используя меры, описанные в этой главе, но изменения в базе данных по-прежнему остаются серьезным препятствием, которое необходимо преодолеть в первую очередь.

11.5.1. Поддержание стабильности схемы базы данных

Чтобы решить описанную проблему, необходимо найти решение, гарантирующее стабильность схемы базы данных. Когда компонент требует изменений в схеме базы данных, прежде всего следует выполнить соответствующие изменения в базе данных. Это делается независимо от развертывания самого компонента. Например, схема базы данных изменяется только в определенные дни недели, тогда как отдельные компоненты могут изменяться в любые дни. Эти изменения можно обезопасить соответствующими тестами и мерами предосторожности.

Если требуется внести изменения в базу данных, сначала производятся эти изменения, и только потом выполняется развертывание компонентов, которые должны работать с измененной базой данных. Часто изменения в базе данных невозможно откатить из-за того, что эта процедура по своей сложности сопоставима с предшествующей процедурой внесения изменений в исходную схему базы данных. Реализация и тестирование этой процедуры требуют больших усилий, поэтому данный шаг часто опускается. Это отличает изменения в базе данных от изменений в компоненте, откатить которые относительно просто. Конечно, это осложняет быстрый ввод изменений в эксплуатацию.

Как показывает практика, прагматичный подход — лучший способ преодоления этой проблемы. Вместо поиска решения проблемы ее следует избегать всеми силами, изменяя схему базы данных как можно реже. Решение проблемы в любом случае окажется более трудоемким, поэтому, хотя данный подход может показаться не очень изящным, он несложен в реализации. Более того, он с успехом используется во многих проектах.

Однако подход, предложенный выше, не укладывается в философию непрерывного развертывания, поскольку фундаментальной целью методологии непрерывного развертывания является максимальная автоматизация процессов и как можно более частое их выполнение. Благодаря этому циклы выпуска с продолжительностью в несколько месяцев замещаются более частыми выпусками, которые порою могут происходить до нескольких раз на день. Тот факт, что цикл выпуска без применения методов непрерывного развертывания занимает так много времени, обусловлен рисками, связанными с классическими подходами, которые приходится смягчать мероприятиями, выполняемыми вручную — в точности как в случае с базами данных.

11.5.2. База данных = компонент

Интерпретация базы данных как простого компонента — еще одно возможное решение проблемы. Если компонент меняет свой интерфейс, он все равно должен продолжать поддерживать старую версию интерфейса. Интерфейсом базы данных является ее схема. Следовательно, схема также должна иметь версии, а изменения в схеме должны производиться так, чтобы обеспечивалась поддержка предыдущей версии. Когда, например, в схему добавляется новый столбец, любые операции чтения могут просто игнорировать дополнительные данные. Для операций записи в базе данных должны быть предусмотрены подходящие значения по умолчанию. Когда в таблицу добавляется новая запись без значения для добавленного столбца, база данных должна подставить значение по умолчанию. Это гарантирует, что компоненты, разработанные под старую версию схемы базы данных, будут корректно работать с новой версией. Если требуется удалить столбец, сначала схему необходимо изменить так, чтобы значения для этого столбца оказались необязательными. Впоследствии может быть развернута новая версия программного обеспечения, которая больше не записывает или не читает этот столбец. Наконец, когда наступит подходящий момент для обновления схемы базы данных, столбец можно удалить.

11.5.3. Представления и хранимые процедуры

Существуют также другие приемы обеспечения совместимости. Например, для одновременной поддержки старой и новой версий схемы можно использовать представления.

Кроме того, схему можно полностью скрыть, разрешив доступ только к хранимым процедурам. В этом случае схема изменяется без всяких ограничений при условии неизменности интерфейса хранимых процедур. В итоге база данных превращается в еще один обычный компонент со своим прикладным интерфейсом в виде комплекса хранимых процедур. В этом случае так называемый уровень хранения реализуется в базе данных. Однако такое решение часто оказывается сложным в реализации.

Однако, несмотря на сложность реализации, данный подход поддерживает обратную совместимость, что для баз данных может оказаться более важным аспектом, чем для других компонентов, поскольку изменения в базах данных осуществляются намного сложнее.

11.5.4. Отдельная база данных для каждого компонента

К слову сказать, проблему можно смягчить, создав для каждого компонента свою базу данных или хотя бы свою схему базы данных. В этом случае изменения в базе данных требуется согласовывать только с одним компонентом — это намного проще и снижает риски, связанные с изменением базы данных.

11.5.5. Базы данных NoSQL

Проблемы со схемами характерны для реляционных баз данных. Базы данных NoSQL более гибкие в отношении схем и могут обрабатывать записи практически с любой структурой. То есть базы данных NoSQL способны одновременно обрабатывать записи с совершенно разными структурами. Конечно, при этом все еще требуется выполнять соответствующие преобразования записей. И правила совместимости продолжают действовать — пока компоненты ожидают получить определенные столбцы, эти столбцы не могут быть удалены из набора данных. Тем не менее базы данных NoSQL получают важное преимущество в сценариях непрерывного развертывания благодаря своей гибкости. Эти преимущества сами по себе могут стать веской причиной для использования базы данных данного типа.

11.6. Микрослужбы

Микрослужбы [8] — не только представляют интересный подход к организации программного обеспечения, которому в последнее время уделяется много внимания, но также обеспечивают более полное соответствие идеям непрерывного развертывания.

Микрослужбы помогают разбить систему на модули. Отличительной чертой этого подхода является возможность развертывания микрослужб независимо друг от друга. В частности, микрослужбы могут быть, например, контейнерами Docker, в каждом из которых установлено программное обеспечение, составляющее микрослужбу. Все вместе микрослужбы образуют законченную программную систему. Они поддерживают разные части веб-интерфейса. То есть система может состоять из микрослужб, обладающих HTML-ссылками друг на друга. Другой вариант: они могут составлять раз-

ные части HTML-страниц, вызываться из JavaScript и таким способом отображаться вместе. Этот подход является основой для создания автономных систем (Self-Contained Systems, SCS) [9].

Микрослужбы также предоставляют услуги, например, через службы REST, которые в свою очередь могут использоваться другими микрослужбами, внешними системами или мобильными клиентами. Кроме того, существуют отдельные микрослужбы, реализующие пользовательский интерфейс и прикладную логику.

11.6.1. Микрослужбы и непрерывное развертывание

Данный архитектурный подход имеет много преимуществ, особенно для непрерывного развертывания.

- Как описывалось в разделе 11.3, деление на несколько независимых единиц развертывания может упростить реализацию конвейера непрерывного развертывания, потому что в этом случае конвейеры получаются менее сложными. Риски, сопутствующие развертыванию, также снижаются, поскольку в процессе развертывания изменяется не вся система, а только одна микрослужба.
- Микрослужбы взаимодействуют друг с другом через соответствующие интерфейсы (раздел 11.3). Поддержка разных версий интерфейсов позволяет организовать независимое развертывание микрослужб. Если возникает необходимость изменить интерфейс, сначала развертывается микрослужба, поддерживающая одновременно новую и старую версии интерфейса. И только после этого развертываются все остальные микрослужбы, поддерживающие новый интерфейс. Затем поддержка старого интерфейса может быть прекращена. Если каждая микрослужба реализует отдельную часть пользовательского веб-интерфейса, такая поддержка версий интерфейсов является исключением, потому что в этом случае микрослужбы практически не взаимодействуют друг с другом, а просто представляют части веб-интерфейса. Соответственно, реализация системы упрощается, поскольку отпадает необходимость в поддержке нескольких версий интерфейсов и исчезают сопутствующие проблемы.
- Микрослужбы должны быть устойчивыми: они должны продолжать работать, когда другие микрослужбы оказываются недоступными. Со-

ответственно, они должны реализовать подход «Страховка от сбоев» и гарантировать устойчивость, как обсуждалось в разделе 11.4. Это способствует снижению рисков при развертывании, потому что остановка отдельной микрослужбы не повлияет на работу других микрослужб.

- В отношении баз данных (раздел 11.5) каждая микрослужба должна иметь собственные данные. То есть микрослужбы не должны иметь общих схем. Это уменьшит проблемы, связанные с обновлением баз данных: при развертывании одной микрослужбы потребуется изменить только одну базу данных. Аналогично каждая микрослужба способна работать со своей отдельной базой данных. Например, микрослужбы могут использовать базы данных NoSQL, если это имеет определенный смысл в конкретном контексте. Естественно, использование базы данных NoSQL единственной микрослужбой менее рискованно, чем преобразование всего приложения.

11.6.2. Внедрение непрерывного развертывания с микрослужбами

Итак, микрослужбы прекрасно поддерживают архитектурные требования непрерывного развертывания. Кроме того, с помощью микрослужб легко можно расширить возможности унаследованной системы. Поэтому, когда реализация конвейера непрерывного развертывания для всей системы оказывается слишком трудоемкой и сложной задачей, ее возможности можно расширить с помощью микрослужб, для которых непрерывное развертывание реализуется намного проще. Достижение целей непрерывного развертывания в сочетании с возможностью расширения унаследованных систем посредством микрослужб для многих проектов оказывается решающей причиной поэтапной замены таких систем микрослужбами. То есть внедрение непрерывного развертывания часто превращается во внедрение микрослужб.

11.6.3. Микрослужбы способствуют внедрению непрерывного развертывания

Однако микрослужбы представляют не только хороший повод для внедрения непрерывного развертывания. Система на основе микрослужб со-

стоит из множества артефактов, развертываемых независимо. Этот подход дает преимущества, только если каждая микрослужба имеет максимально автоматизированный конвейер непрерывного развертывания. Из-за большого количества развертываемых артефактов ручное управление становится практически невозможным ввиду большой трудоемкости. В аспекте развертывания и эксплуатации, микрослужбы представляют собой самые большие проблемы. Поэтому реализация непрерывного развертывания становится практически неизбежной при использовании микрослужб. Фактически архитектуру микрослужб нельзя реализовать без реализации непрерывного развертывания.

11.6.4. Организация

Организационные структуры, представленные в разделе 11.2, применимы не только к непрерывному развертыванию, но и к микрослужбам: они также поддерживают индивидуальную ответственность и самоорганизацию. Микрослужбы обеспечивают свободу выбора технологий: они могут быть реализованы на разных языках программирования и на разных платформах. Коллектив, отвечающий за развитие конкретной микрослужбы, практически не должен координировать свою работу с другими коллективами. Это позволяет экономить на усилиях по координации, потому что каждый коллектив может принимать свои технологические решения и упрощать себе работу с разными прикладными аспектами. Кроме того, отдельные конвейеры непрерывного развертывания обеспечивают полную независимость в отношении внедрения программного обеспечения в эксплуатацию. Коллективы, занимающиеся реализацией отдельных функциональных особенностей, также получают выгоды, потому что не только имеют возможность вести разработку независимо, но также могут независимо друг от друга внедрять изменения в рабочее окружение.

Таким образом, микрослужбы могут стать прекрасным дополнением для непрерывного развертывания.

11.7. Внедрение новых возможностей

Новые возможности обычно реализуются в программном коде и становятся доступными сразу после завершения реализации и развертывания в рабочем окружении.

11.7.1. Отдельные ветви для новых возможностей

Чтобы изолировать разработку новых возможностей от разработки всей системы, в некоторых проектах создаются отдельные ветви в системе управления версиями. То есть для новой возможности создается отдельная ветвь, в которую вносятся все изменения, связанные с этим. Разработка других возможностей или исправление ошибок осуществляется в других ветвях, поэтому изменения не влияют друг на друга. Такой подход позволяет полностью изолировать реализацию разных возможностей, чтобы дать возможность разным коллективам параллельно решать разные задачи. Кроме того, новые возможности, разрабатываемые отдельно друг от друга, также могут вводиться в эксплуатацию по отдельности.

Этот подход несовместим со стратегией непрерывного развертывания. По идее, для каждой ветви должен настраиваться отдельный конвейер непрерывного развертывания. Однако это слишком трудоемко. Кроме того, в этом случае остается недостижимой фундаментальная цель непрерывного развертывания: ветви развиваются параллельно и в какой-то момент времени должны объединиться. Проблемы, обусловленные несовместимостью изменений в разных ветвях, проявляются только на этапе их объединения, в то время как главная задача конвейера непрерывного развертывания — обеспечить постоянное и максимально быстрое получение обратной связи. В случае ветвления получение обратной связи о возможных проблемах несовместимости откладывается до момента слияния ветвей. По этой причине прием создания отдельных ветвей для разработки новых возможностей плохо укладывается в идеологию непрерывного развертывания.

Следовательно, все разработчики должны работать в одной, общей ветви. Однако и это решение не избавляет от проблем, связанных с реализацией новых возможностей: должен иметься какой-то механизм активации новых возможностей в рабочем окружении. В конце концов, не всякую новую возможность можно реализовать целиком и сразу, поэтому возможность в незаконченном состоянии не должна быть доступна пользователям. Кроме того, активация новых возможностей зачастую тесно связана с бизнес-решениями и может быть приурочена, например, к рекламной кампании, проводящейся в определенное время.

11.7.2. Переключение возможностей

Прием переключения возможностей позволяет совместно разрабатывать и непрерывно интегрировать новые особенности, но активировать их толь-

ко в определенный момент времени. Суть этого приема заключается в реализации выключателей для активации или деактивации определенных возможностей. Благодаря такому подходу существует возможность вести разработку новой особенности и постоянно переносить код в рабочее окружение, оставляя его неактивным.

11.7.3. Преимущества

Этот подход имеет целый ряд преимуществ.

- Реализация отделена от развертывания. Код реализации может переноситься в рабочее окружение до активации новой возможности. Это позволяет коллективам соблюдать сроки разработки и сохранять более спокойный ритм работы: обычно новая возможность должна быть доступна пользователям не раньше установленного срока. При обычном стиле разработки код должен вводиться в эксплуатацию точно в этот момент времени — не раньше и не позже. Используя прием переключения возможностей, код можно переносить в рабочее окружение до назначенной даты, а в нужный срок просто активировать новую функцию с относительно низкими рисками.
- Новую возможность можно также предварительно протестировать. Для этого достаточно активировать ее для определенных пользователей. С помощью таких специализированных учетных записей можно проверить, действительно ли новая возможность действует так, как предполагается. Для этого не требуется создавать отдельное тестовое окружение, а убедиться в работоспособности новой функции можно прямо в рабочем окружении. Разумеется, необходимо принять все меры предосторожности, чтобы изолировать тестовые операции и избежать, например, фактических поставок товаров по пробным заказам. С другой стороны, нет необходимости создавать тестовое окружение, идентичное рабочему. Обычно подобное требование просто невыполнимо из-за слишком высоких затрат и потому, что внешние системы не могут быть представлены несколькими версиями.
- Переключение возможностей также позволяет проверить реакцию пользователей или выборочно предоставить некоторые возможности определенным группам пользователей. В этом случае возможность должна быть активирована для определенного круга пользователей и оставаться недоступной для других. Впоследствии можно проверить, действительно ли новая возможность дает предполагаемый положи-

тельный эффект, например способствует увеличению объема продаж или других бизнес-показателей. На техническом языке это называется А/В-тестированием. С прикладной точки зрения А/В-тестирование играет очень важную роль, поскольку, как показывает опыт, многие изменения в программном обеспечении не дают ожидаемого эффекта, а порою даже оказывают отрицательное влияние. А/В-тестирование позволяет на ранних этапах выявить негативные последствия и сосредоточиться на изменениях, способствующих развитию бизнеса — например, способствующих росту объема продаж.

- Кроме того, новую возможность можно сначала активировать на одном или нескольких серверах. Если реализация содержит ошибку, например, вызывающую крах системы, ее влияние можно ограничить небольшим количеством серверов. Постепенно новую возможность можно активировать на все большем количестве серверов. Этот прием называется канареечным развертыванием (раздел 7.5).
- Наконец, переключение возможностей может пригодиться за рамками методологии непрерывного развертывания. Например, некоторые возможности можно отключать на периоды недоступности некоторой системы. Эти возможности станут недоступны, но это может предотвратить необходимость остановки всей системы в целом. Но это в большей мере относится к приложению, а не к непрерывному развертыванию.

11.7.4. Примеры использования переключения возможностей

Переключение возможностей [10] имеет много преимуществ. Можно выделить три основные области применения этого приема.

- Переключение для выпуска: цель — отвязать активацию возможности от даты выпуска, чтобы позволить поэтапно переносить код в рабочее окружение. Сначала производится постепенное развертывание кода, пока возможность отключена. А по завершении реализации и тестирования она вводится в эксплуатацию простой операцией активации.
- Переключение для поддержки бизнеса: цель — проверить реакцию пользователей или выборочно предложить некоторые возможности определенным группам пользователей.
- Переключение для нужд эксплуатации: цель — позволить деактивировать возможности, чтобы избежать остановки всего приложения.

Каждый из этих сценариев имеет свои отличия и, соответственно, свои требования к механизму переключения.

11.7.5. Недостатки

Прием переключения возможностей имеет также недостатки: он увеличивает сложность программного обеспечения, поскольку появляется необходимость различать активное и неактивное состояния. Кроме того, помимо фактической реализации необходимо иметь тесты.

Для уменьшения сложности предпочтительнее реализовать переключение возможностей как можно проще. Иногда, например, достаточно запретить отображение ссылки на веб-странице.

Кроме того, количество переключателей желательно ограничить. В редких случаях достаточно реализовать единственный переключатель, активирующий или деактивирующий все новые возможности. С другой стороны, можно реализовать более дробное переключение. Выбор варианта во многом зависит от целей, для которых реализуется переключение возможностей. Если, например, предполагается провести А/В-тестирование некоторых возможностей, переключение должно быть реализовано для каждой возможности в отдельности. То же верно для ситуации, когда переключение используется для компенсации недоступности каких-то систем. В зависимости от фактических особенностей использования переключателей также следует протестировать различные их комбинации. Когда некоторые возможности уже протестированы, но еще не активированы в рабочем окружении, необходимо проверить конфигурацию переключателей в режиме эксплуатации. Иначе высока вероятность того, что изменения в коде создадут проблемы, проявляющиеся только в процессе эксплуатации и только с данной конфигурацией.

Переключатели, ставшие ненужными, должны удаляться из кода, чтобы уменьшить его сложность.

11.8. В заключение

Методология непрерывного развертывания оказывает влияние на архитектуру программного обеспечения, даже если это влияние неочевидно. Поэтому архитектуру программного обеспечения в обязательном порядке

следует адаптировать при внедрении непрерывного развертывания. Недостаточно просто изменить процессы. Этот аспект часто игнорируется и может препятствовать успешной реализации непрерывного развертывания. Помимо классических нефункциональных требований (таких, как производительность или масштабирование), правильный подход к внедрению и развертыванию новых возможностей также является важным фактором влияния на архитектуру, структуру компонентов и выбор технологий.

ПРОБУЙТЕ И ЭКСПЕРИМЕНТИРУЙТЕ

- ▲ Проведите исследования по теме «Микрослужбы» и оцените их влияние на реализацию непрерывного развертывания.
- ▲ Какие библиотеки для реализации приема переключения возможностей доступны в вашем языке программирования? Какими преимуществами они обладают? Для Java, например, есть библиотека Tooglz [11].

Выберите проект, с которым вы хорошо знакомы.

- ▲ Какие компоненты вы можете выявить в проекте? Помощь в этом вам окажут документы с описанием архитектуры или обзорные презентации.
- ▲ Какую технологию (например, REST) можно использовать для ослабления связей между компонентами с целью упростить реализацию непрерывного развертывания? Какие преимущества имеет эта технология по сравнению с другими подходами?
- ▲ Какие компоненты следует изолировать таким способом? Обратите внимание на увеличение затрат, связанных с реализацией такой изоляции.
- ▲ Как должны обрабатываться изменения в схеме базы данных? Выберите одну из предложенных стратегий.

Ссылки

1. https://ru.wikipedia.org/wiki/Архитектура_программного_обеспечения
2. <https://ru.wikipedia.org/wiki/REST>

3. <https://ru.wikipedia.org/wiki/JSON>
4. <https://developers.google.com/protocol-buffers/>
5. <http://www.w3.org/XML/>
6. https://en.wikipedia.org/wiki/Robustness_principle и <http://tools.ietf.org/html/rfc793#section-2.10>
7. Michael T. Nygard: *Release It!: Design and Deploy Production-Ready Software, Pragmatic Programmers, 2007, ISBN 978-0-97873-921-8* (Майкл Хейзгард. Release it! Проектирование и дизайн ПО для тех, кому не всё равно. СПб.: Питер, 2015. ISBN: 978-5-496-01611-7. — Примеч. пер.).
8. Eberhard Wolff: *Microservices: Flexible Software Architecture, Addison Wesley, 2016, ISBN 978-0-13460-241-7*.
9. <http://scs-architecture.org/>
10. <https://martinfowler.com/bliki/FeatureToggle.html>
11. <https://www.togglz.org/>

12

Заключение: основные преимущества

Задача этой книги состояла в том, чтобы показать, что непрерывное развертывание — это намного больше, чем простая автоматизация инфраструктуры и оптимизация развертывания. В действительности главными целями непрерывного развертывания являются следующие.

○ Обратная связь

Более быстрое и частое развертывание позволяет быстрее получать обратную связь из рабочего окружения. Различные этапы тестирования также имеют целью как можно быстрее сообщить о проблемах в программном обеспечении.

○ Автоматизация

Автоматизация тестирования и развертывания позволяет получать обратную связь практически мгновенно.

○ Воспроизводимость

Повсеместная автоматизация позволяет точно воспроизводить результаты тестирования и установки программного обеспечения.

○ Простота поиска ошибок

Все изменения в настройках серверов и в программном обеспечении версионизируются. Изменения в настройках брандмауэра, например, можно быстро отменить, просто вернув на место прежнюю версию конфигурации. Кроме того, всегда можно четко определить, какие изменения были выполнены не только в программном обеспечении, но и в инфраструктуре.

В конечном итоге непрерывное развертывание является естественным продолжением непрерывной интеграции. Все изменения в программном обеспечении не только непрерывно интегрируются, но также производится всестороннее их тестирование и — при соответствии критериям качества — внедрение в эксплуатацию.

Более высокая скорость реализации изменений превращается в дополнительные преимущества для предприятия. А более высокая надежность особенно привлекательна для IT-подразделений, поскольку способствует уменьшению психологического напряжения.

Методология непрерывной интеграции давно стала стандартным явлением в промышленности. Весьма вероятно, что методологию непрерывного развертывания ожидает такой же успех и в будущем она превратится в привычный процесс разработки программного обеспечения. Однако непрерывное развертывание влияет на весь процесс вплоть до эксплуатации и, соответственно, представляет собой более фундаментальное изменение, чем непрерывная интеграция.

Следующим шагом после внедрения непрерывного развертывания, вероятно, станет изменение архитектуры программного обеспечения, как обсуждалось в главе 11 «Непрерывное развертывание, DevOps и архитектура ПО», чтобы упростить реализацию непрерывного развертывания. Архитектура на основе микрослужб [1] очень хорошо подходит для непрерывного развертывания, она позволяет уменьшить размеры компонентов, что в свою очередь ведет к упрощению конвейера.

С организационной точки зрения непрерывное развертывание тесно связано с интеграцией DevOps. Однако обратная связь с эксплуатацией и определение новых возможностей также представляют ценность для предприятия и таких подразделений, как отделы продаж и маркетинга. Здесь в игру вступают методики Lean Startup [2] и Design Thinking [3] (см. разделы 1.4.7 и 10.5). То есть непрерывное развертывание — это методика оптимизации разработки программного обеспечения, которая дает огромные преимущества. Однако нет предела совершенству.

Ссылки

1. Eberhard Wolff: *Microservices: Flexible Software Architecture*, Addison Wesley, 2016, ISBN 978-0-13460-241-7.
2. Eric Ries: *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Business, 2010, ISBN 978-0-67092-160-7 (*Эрик Рис. Бизнес с нуля. Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели. М.: Альпина Паблишер, 2016. ISBN: 978-5-9614-6028-5. — Примеч. пер.*).
3. https://en.wikipedia.org/wiki/Design_thinking

Эберхард Вольф

Continuous delivery. Практика непрерывных апдейтов

Перевел с английского А. Киселев

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Римицан</i>
Литературный редактор	<i>Е. Пасечник</i>
Художественный редактор	<i>С. Заматевская</i>
Корректоры	<i>Н. Витько, И. Тимофеева</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 191123, Россия, город Санкт-Петербург,
улица Радищева, дом 39, корпус Д, офис 415. Тел.: +78127037373.

Дата изготовления: 08.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 27.07.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 25,800. Тираж 1500. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электrozаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
Полная информация здесь: <http://www.piter.com/page/avtoru>

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;
e-mail: books@piter.com





Вопросы по продаже электронных книг: тел.: (812) 703-73-74, доб. 6217;
e-mail: kuznetsov@piter.com



**ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:**

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Kiwi-кошелек.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Псылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com
Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com